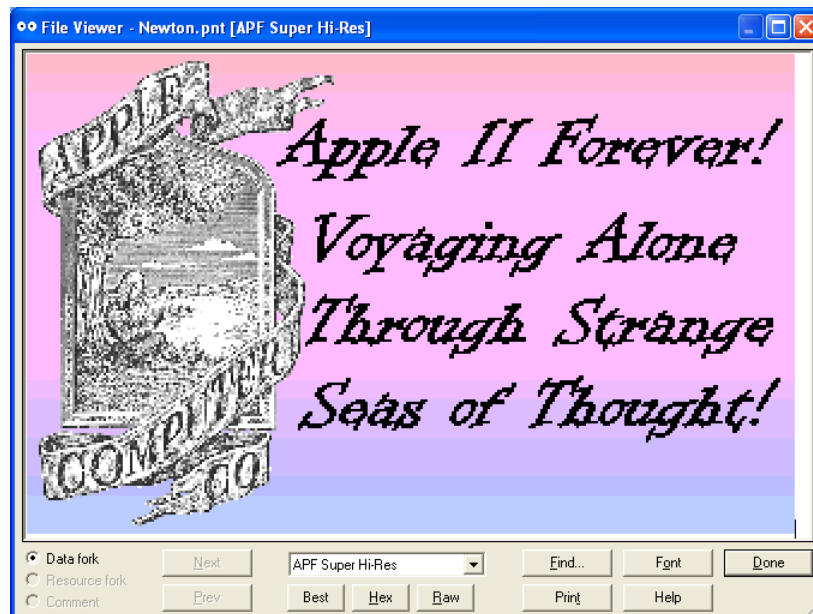


# “Hacking it In” And “UnPacking It Out”



## Portable HackBytes in C

A Case Study: Cross-Platform Development for the Apple IIgs  
By: Bill Buckels © Copyright 2014. All Rights Reserved.  
Research by: Charlie

Andy McFadden  
Antoine Vignau and Olivier Zardini  
STYNX (Jonas Grönhagen)  
Alex Lee  
Rob

ImageMagick Palettization by: STYNX (Jonas Grönhagen)  
Not endorsed by: FaddenSoft LLC and Andy McFadden  
Brutal Deluxe Software (Antoine and Olivier)  
Especially Not endorsed by: Apple Computer and APDA

## **Portable HackBytes – Hacking it In and UnPacking It Out**

### **Table of Contents**

<a href="#">Portable HackBytes – Hacking it In and UnPacking It Out.....</a>	<a href="#">2</a>
<a href="#">Table of Contents.....</a>	<a href="#">2</a>
<a href="#">Forward.....</a>	<a href="#">3</a>
<a href="#">Licence Agreement and Disclaimer.....</a>	<a href="#">5</a>
<a href="#">UnPackBytes() Licence.....</a>	<a href="#">7</a>
<a href="#">Endorsement Disclaimer.....</a>	<a href="#">7</a>
<a href="#">Acknowledgment.....</a>	<a href="#">8</a>
<a href="#">Introduction.....</a>	<a href="#">9</a>
<a href="#">Input/Output File Extension - SHR#C10000 - Pic.....</a>	<a href="#">10</a>
<a href="#">Input/Output File Extension - SH3#C10002 - Brooks.....</a>	<a href="#">11</a>
<a href="#">Input/Output File Extension - PAK#C00001 – Eagle/PackBytes.....</a>	<a href="#">11</a>
<a href="#">PAK versus APF.....</a>	<a href="#">12</a>
<a href="#">No Interpretation Needed .....</a>	<a href="#">12</a>
<a href="#">Less Disk Space.....</a>	<a href="#">12</a>
<a href="#">Better Optimization.....</a>	<a href="#">12</a>
<a href="#">Input/Output File Extension - PNT#C00002 - APF.....</a>	<a href="#">13</a>
<a href="#">Input/Output File Extension – PA3#C00004 – Brooks Eagle/PackBytes.....</a>	<a href="#">15</a>
<a href="#">Additional SHR File Format Notes.....</a>	<a href="#">16</a>
<a href="#">P2P – Pic to Pnt Portable HackBytes Example Program.....</a>	<a href="#">17</a>
<a href="#">Program Organization.....</a>	<a href="#">18</a>
<a href="#">The HackBytes() Function.....</a>	<a href="#">19</a>
<a href="#">HackQuads and the Mid-Stream Encoder.....</a>	<a href="#">27</a>
<a href="#">HackBytes and the Main Stream Encoder.....</a>	<a href="#">28</a>
<a href="#">Portable HackBytes() Optimizations Table.....</a>	<a href="#">29</a>
<a href="#">The HackQuads() Function .....</a>	<a href="#">31</a>
<a href="#">Encoding 256 Color PAK Files with “Dry-Runs” and “Wet-Runs”.....</a>	<a href="#">34</a>
<a href="#">Encoding 3200 Color Brooks PAK Files.....</a>	<a href="#">36</a>
<a href="#">Encoding APF Files.....</a>	<a href="#">39</a>
<a href="#">Transforming “raw” SHR Settings to APF File Settings.....</a>	<a href="#">41</a>
<a href="#">Little Endian and Big Endian Helper Functions.....</a>	<a href="#">42</a>
<a href="#">Brooks Palette to APF Palette Helper Function .....</a>	<a href="#">44</a>
<a href="#">The UnPackBytes() Function.....</a>	<a href="#">45</a>
<a href="#">Decoding Packed SHR Files with UnPackBytes.....</a>	<a href="#">48</a>
<a href="#">Test Results - Compatibility And Regression.....</a>	<a href="#">58</a>
<a href="#">Test Results - Performance – Comparing Apples to Bananas.....</a>	<a href="#">60</a>
<a href="#">Test Results - Mode3200 Files.....</a>	<a href="#">60</a>
<a href="#">The Contenders - Not Your Father’s SHR Converter.....</a>	<a href="#">61</a>
<a href="#">Test Results – Mode320 Files.....</a>	<a href="#">61</a>
<a href="#">Comparing Apples to Apples.....</a>	<a href="#">63</a>

## **Forward**

This article provides a working program called “p2p” as a practical and portable C language programming example of using Apple IIgs “PackBytes” Run-Length Encoding (RLE).

Also provided in this article is a relatively detailed description of Portable HackBytes. This is the PackBytes encoder that I have written, and that I demonstrate in the p2p program and explain in this document.

\*\*\*

PackBytes and UnPackBytes are data compression and unpacking routines native to the Apple IIgs and built-in to the Apple IIgs System Software. If you wish to write programs that do not use the native Apple IIgs routines to support the PackBytes RLE , you need to include a PackBytes encoder and/or an UnPackBytes decoder in your program.

PackBytes and UnPackBytes are built-in to standard Apple IIgs device-dependent graphics file formats targeted at the Apple IIgs Super-Hi Resolution (SHR) display, like the Apple Preferred Format (APF) and the Eagle/PackBytes (PAK) format (both demonstrated in p2p), but other encoded or compressed graphics file formats of the day like PCX and GIF were much more widely used in the wild for lossless graphics interchange. On computers like the IBM-PC, since it has never made much sense to store graphics in Apple IIgs device-dependent formats, like APF or Eagle/PackBytes, which use the PackBytes RLE, the C language source code for a PackBytes encoder seems non-existent.

However, thanks to Andy McFadden, the author of CiderPress, C++ source code for CiderPress’s UnPackBytes has been available for quite some time outside the Apple IIgs (and is modified for use in p2p).

\*\*\*

Portable HackBytes achieves slightly better compression than Apple Computer’s PackBytes encoder by using a simple list processor and simple logic.

Images encoded using either encoder both decode with the same decoders.

In [Lesson 2](#) of his tutorial “[Hacking Data Compression](#)”, Andy McFadden explains how the Apple IIgs PackBytes encoder works. The way Portable HackBytes encodes repeats is different than in Andy’s explanation. Perhaps the way Portable HackBytes works in its entirety is different, but I can’t really tell from Andy’s explanation.

I have never used, nor have I done any analysis on Apple’s implementation of their PackBytes encoder, except to read Andy’s explanation and to compare the size of files

produced by Apple Computer's PackBytes encoder to the smaller size of equivalent files produced by Portable HackBytes.

After reading Andy's explanation, I decided that UnPackBytes wouldn't care how the encoder worked as long as everything decoded properly, and a PackBytes encoder could be both fearless and portable. So I wrote Portable HackBytes, in its entirety using common sense (fundamentally based on Andy McFadden's comments in the CiderPress PackBytes decoder combined with my own brute-force logic).

\*\*\*

I got interested in SHR images almost a year ago. With the help of several other Apple II enthusiasts in the comp.sys.csa2 Internet News Group, I gathered all the historical information I could "get my hands-on" about SHR. I quickly learned that, while several SHR formats existed, the two "raw" screen-size formats ("PIC" and "Brooks") and the "APF" format provide support for almost everything that SHR graphics can be used for. So I set-out to write loaders and conversion utilities in an effort to achieve a reasonably competent knowledge level about using these in my Apple II programming.

The first thing I did was write two 8-bit loaders in Aztec C65; one for "raw" mode320 and mode640 "PIC" files, and another for the equivalent APF files. But to this day I have yet to complete a loader for "Brooks" mode3200 files (a feat not as easily accomplished as loaders for the other two).

For my APF loader noted above, I was able to write UnPackBytes easily by adapting Andy McFadden's C++ UnPackBytes from CiderPress.

When it came to writing my SHR converters, I stayed away from the APF format, and stuck with "raw" format output, because I had not yet put the effort into writing a PackBytes encoder (a feat not as easily accomplished as plagiarizing UnPackBytes from Andy's C++ code).

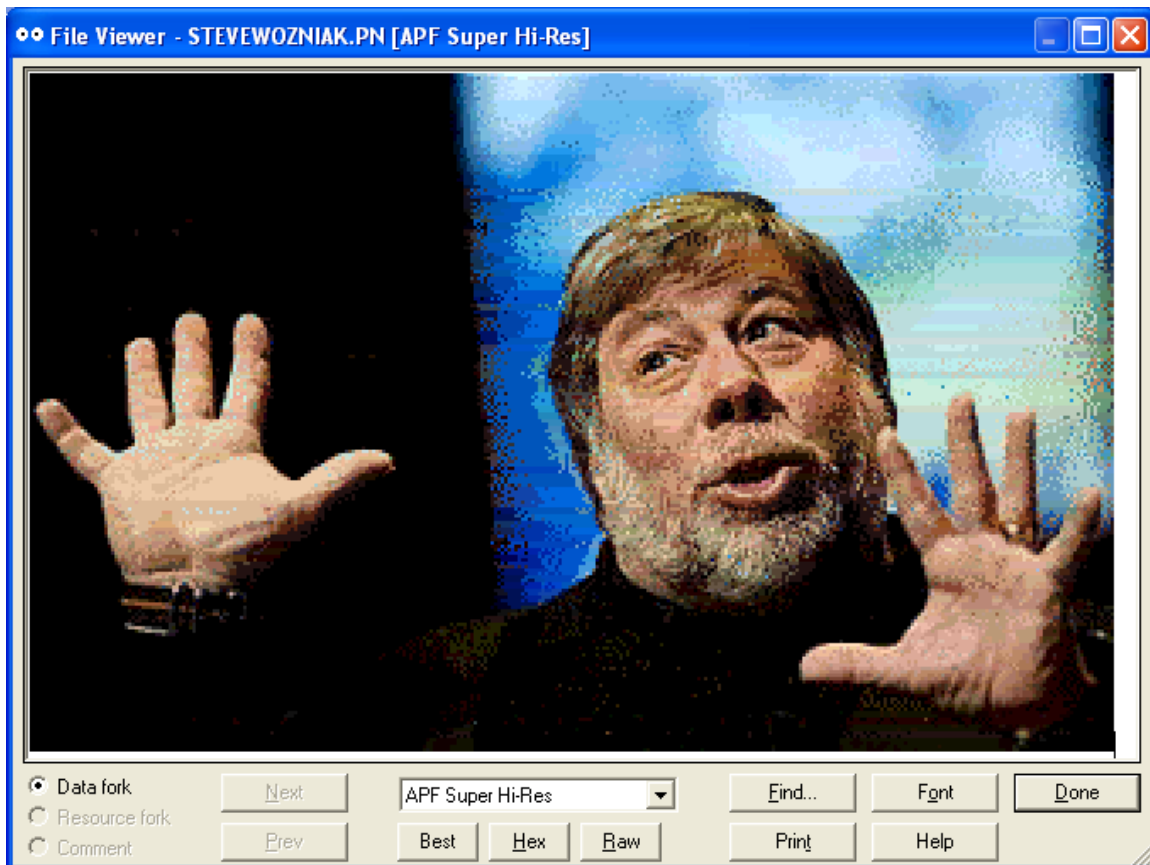
I have yet to run across any C or C++ source to encode to APF or Eagle/PackBytes from "raw" format SHR files. It seems to me that, in the absence of re-usable PackBytes code in the C or C++ language, any APF or Eagle/PackBytes files that I have collected were probably produced on the IIgs where PackBytes is very much a part of the system.

A few weeks back I decided to write a Portable PackBytes encoder in the C programming language, both for the fun of it, but also to pass-on to other Apple II enthusiasts, whether their desktop machines are Windows, Mac, or Linux computers. So after finishing the Portable HackBytes encoder to a workable degree, with better compression than Apple Computer's PackBytes, I am writing this article; to share what I have learned through the experience.

## Licence Agreement and Disclaimer

**Portable HackBytes © and p2p © Copyright Bill Buckels 2014.  
All Rights Reserved.**

**UnPackBytes() Copyright © 2007, FaddenSoft, LLC.  
All rights reserved.**



Under the Conditions stated in the next sections of this document, you have a royalty-free right to use, modify, reproduce and distribute the p2p program, including source code, documentation, and the other baggage it comes with in any way whatsoever. Neither Bill Buckels nor Andy McFadden has any warranty, liability, or endorsement obligations resulting from said use and distribution in any way.

- Andy McFadden requests attribution for the use of UnPackBytes().
- The rest of the p2p program including Portable HackBytes needs no attribution.

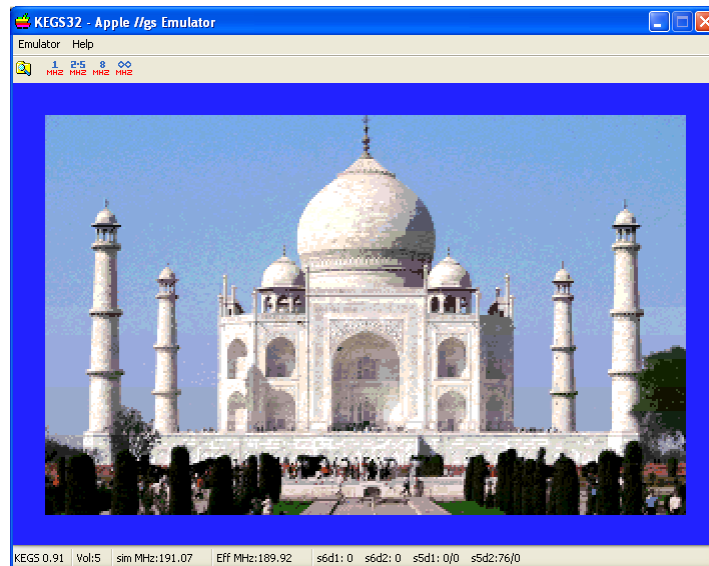
Please read the next sections carefully.

**UnPackBytes() Copyright (c) 2007, FaddenSoft, LLC. All rights reserved.**

The UnPackBytes() function in the p2p program is taken literally from CiderPress (ReformatBase.cpp). It is “dummied-down to” C from C++ and lightly modified, but by itself likely does not merit consideration as a derivative work as used in the p2p program. The p2p program itself is more like a derivative work (of CiderPress), but the UnPackBytes() function itself likely does not merit consideration as a derivative work at all, since its use in both CiderPress and p2p is not transformative, when compared to Apple Computer’s UnPackBytes Apple IIGs ToolBox routine.

It would be a different story if the UnPackBytes() function in p2p was written before the UnPackBytes Apple IIGs ToolBox routine. And it would be a different story if p2p used the UnPackBytes() function in a significantly different manner from the general and historical use of Apple Computer’s UnPackBytes().

The author of an original work or a derivative work has rights called “Copyright”. Since I did not write UnPackBytes and my use of UnPackBytes is not significantly transformative, I cannot grant permission for use of UnPackBytes. Also I don’t honestly know if Andy’s Copyright extends to UnPackBytes or if Apple Computer’s Copyright extends to UnPackBytes but I suspect that Apple Computer is the Copyright Holder. So as you can see, this is quite a dilemma that seemingly does not stop. What I do know for sure that it is both customary and a matter of good manners and etiquette to provide attribution and to respect the licences of other software authors like Andy McFadden when you use their code in your code.



*The image above compressed by p2p using Portable HackBytes shows the users of a Work enjoying access privileges.*

## UnPackBytes() Licence

Copyright © 2007, FaddenSoft, LLC  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, is permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of FaddenSoft, LLC nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

**THIS SOFTWARE IS PROVIDED BY FaddenSoft, LLC "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.**

**IN NO EVENT SHALL FaddenSoft, LLC BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.**

### Endorsement Disclaimer

None of any of my work is endorsed by Andy McFadden. So “be very afraid”!

## Acknowledgment

*The 256 color APF file below converted by p2p using Portable HackBytes shows a recent meeting of the French Apple IIgs Users Group:*



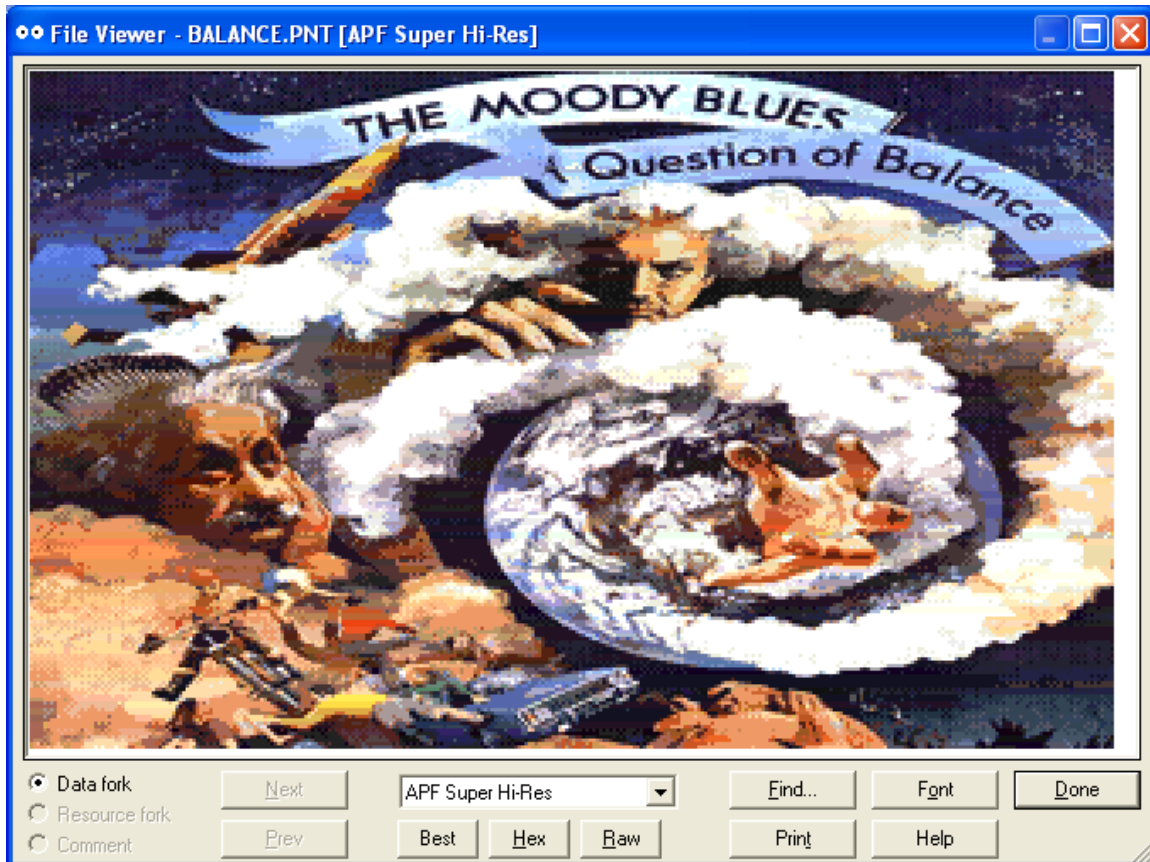
First off this is not an endorsement of my work by Brutal Deluxe Software, so “be very afraid”!

It is impossible to write graphics converters for the Apple IIgs and not be affected by the brilliant and unparalleled work of Antoine Vignau and Olivier Zardini of Brutal Deluxe Software. SHR Graphics on the Apple IIgs would not even still be alive without Brutal Deluxe, and not nearly as alive in the first place. Right from the beginning of my Apple II graphics development adventure in recent years, Antoine has been constant, always ready with advice and pointing me to resources. When I advanced to IIgs graphics development, Olivier Zardini also provided significant advice.

And for that, Brutal Deluxe Software gets special thanks, along with Andy McFadden and STYNX and Charlie, for making Portable HackBytes possible.

Just for the record!





## **Introduction**

According to Apple Computer, there are 2 ProDOS SHR FileTypes (used primarily on the Apple IIgs). The p2p program and its Portable HackBytes encoder rolls with that:

- FileType \$C1 - PIC – raw data
- FileType \$C0 – PNT – transformed data

PIC files store their data in “raw” format. PNT files are SHR files produced by Paint Programs and other programs with the savvy to transform and encode and decode them.

The Apple Preferred Format (APF) file and the Eagle/PackBytes file used primarily on the Apple IIgs are PNT files. They store run-length encoded device dependent bitmapped graphics images for display in Apple IIgs Super-Hi Res (SHR) mode320, mode640 and in the case of APF, mode3200.

- The scan-lines in an APF file are not stored in “raw” format... each scan-line is Run Length Encoded (RLE) individually using the Apple IIgs PackBytes RLE format. Images stored in APF can be wider and longer than the screen.

- The entire Eagle/PackBytes file is encoded using PackBytes and since it is nothing more than a PackBytes encoded file of SHR screen memory, it is confined to screen resolution.

For the purposes of this article, and for most “practical” purposes (if indeed any practical purposes exist today for Apple IIs SHR graphics), I have confined my examples to creating PNT files from screen-sized “raw” SHR images in the 3 graphics modes supported by the APF file format.

The p2p program converts between PNT and PIC file formats in SHR screen resolution only. For conversion to PNT format it uses Portable HackBytes and for conversion to PIC format it uses UnPackBytes() adapted from CiderPress. For the purpose of SHR file interchange using the p2p program, only 4 official SHR FileTypes and one unofficial File Type are supported:

SCx Types	Input File Any Name	Output File Automatic Extension
<b>\$C0 PNT</b>	<b>Apple IIs Packed Super HiRes</b>	
<b>\$0000</b>	<b>Paintworks Packed Super Hi-Res</b>	<b>Not Supported</b>
<b>\$0001</b>	Packed Super HiRes	<b>\$C1 \$0000 SHR</b>
<b>\$0002</b>	Apple Preferred Format	<b>\$C1 \$0000 SHR</b> <b>\$C1 \$0002 SH3</b>
<b>\$0003</b>	<b>Packed QuickDraw II PICT</b>	<b>Not Supported</b>
<b>\$0004</b>	Packed Super HiRes 3200 “Brooks”	<b>\$C1 \$0002 SH3</b>
<b>\$C1 PIC</b>	<b>Apple IIs Super HiRes</b>	
<b>\$0000</b>	Super Hi-Res Screen Image	<b>\$C0 \$0001 PAK</b> <b>\$C0 \$0002 PNT</b>
<b>\$0001</b>	<b>QuickDraw PICT</b>	<b>Not Supported</b>
<b>\$0002</b>	Super HiRes 3200 “Brooks”	<b>\$C0 \$0002 PNT</b> <b>\$C0 \$0004 PA3</b>

### Input/Output File Extension - SHR#C10000 - Pic

```

/* FileType $C1 AuxType $0000 - mode320 and mode640 */
typedef struct tagPICFILE
{
    uchar line[200][160]; /* 3200 bytes */
    uchar scb[200];
    uchar padding[56];
    uchar pal[16][32];
} PICFILE;

```

The PIC File Aux Type 0000 is easiest and quickest to load. It is a RAW BSAVED image of SHR memory, which includes 3200 bytes of screen memory of 200 x 160 byte scanlines, followed by 200 scanline control bytes padded to 256 bytes, followed by 512 bytes of 16 palettes of 16 12 bit colors in an array of Motorola \$0RGB Words.

Because the standard SHR Screen Memory is in two resolutions, mode320 and mode640, which are selected by soft switches as part of the Screen Memory, this file format supports both resolutions, but does not support images with higher resolutions than the SHR screen. Even though this file is the lowest common denominator for SHR and easiest to load, a specific load sequence including control over main and auxiliary memory is still required, so something like intermediate programming skills are required for this file to be of use by a programmer.

### **Input/Output File Extension - SH3#C10002 - Brooks**

```
/* FileType $C1 AuxType $0002 - mode3200 */
typedef struct tagBROOKSFILE
{
    uchar line[200][160];
    uchar pal[200][32];
    /* $ORGB table buffer palette entries 0-16 reversed */
} BROOKSFILE;
```

The PIC File Aux Type 0002 (Brooks Format) is quick to load but not easy for a programmer to display. Displaying this type of file requires that the program code is constantly synching with internal screen updates to constantly move palettes in Real-time from a buffer of 200 palettes into the 16 palettes supported by SHR. This leaves little time for anything else, and less time on a slower Apple IIgs, and even less time on an Apple IIe with an Apple Video Overlay Card (VOC) but no accelerator.

*While Todd Whitesel has written a Brooks Loader for the VOC on an Apple IIgs, I have never seen a VOC Brooks loader for the Apple IIe. I don't know whether an 8 bit program like SHR View would work for Brooks on a IIe equipped with a VOC since I don't have a VOC. Todd's file format for the VOC's standard interlace mode(from an earlier effort) is also not as robust as my own, since it only allows half the palettes.*

*I am speculating that he likely based his format on what was easiest for him to create in whatever IIgs Paint program he used at the time. I also have no evidence of any Brooks format files that follow his HT and HB formats. While my BMP2SHR utility supports the creation of Brooks Files for the VOC's interlace mode, in practice these may be quite useless. P2p does not support those, or any provision for the VOC's interlace mode400 at all, simply because I did not want to do the extra programming needed to convert "raw" file pairs to "stretched" APF files of screen width, but with 400 lines.*

### **Input/Output File Extension - PAK#C00001 – Eagle/PackBytes**

The least elegant but the most efficient of the PNT format files supported by p2p is the \$C0, \$0001 Packed Super HiRes. It is just an entire PIC file, encoded using PackBytes encoding from start to finish. It is not "officially" extensible to BROOKS, and it is not officially extensible to sizes smaller or larger than the SHR screen.

## **PAK versus APF**

A Packed Super Hi Res file offers several advantages over a Screen Size APF:

### **No Interpretation Needed**

- PAK files do not need to be interpreted. In almost the same way as a PIC file is loaded, a PAK file is unpacked using UnPackBytes() directly into screen memory as a “blob”. No further work required.
- Apple would probably not call Eagle/PackBytes “flexible” since it is consistently simple. In fact, it is so simple that programs like Activision’s PaintWorks may have decided not to support this format because it is too in-flexibly inelegant to flexibly contain inflexible baggage like QuickDraw stuff not to be found elsewhere on our flexible planet.

### **Less Disk Space**

A PAK file generally takes up less space on a disk, since it hasn’t got the overhead of “real” header like an APF.

### **Better Optimization**

- During the development of Portable HackBytes, I decided to try encoding these using both blob and line segment techniques to see which method was the most optimal. Because a PAK file is not scanline oriented, it can be encoded in segments, by lines, or as one big blob, with varying results for every file. The most optimal method is used “automagically” for the final encoding.
- You will see this optimization method in the p2p code later in this document. By comparison, the restriction of a Screen Size APF to 160 byte scanline encoding often (not always) restricts compression. On the other hand, sometimes repeats align on scanlines, so sometimes encoding as a blob restricts compression (unless comparisons between the two are used).
- When I ran tests using alignment on 256 byte sectors and 512 byte blocks, compression was almost always worse, but using segmentation based on scanline boundaries and iteratively and repetively encoding and comparing variable segments often really shaves-off the fat.

Comparative results between Apple’s PackBytes encoder and mine are shown later in this document. More could likely be done with Portable HackBytes to reduce the size further by additional segmentation options combined with more iterative techniques; p2p just explores some of these.

## Input/Output File Extension - PNT#C00002 - APF

APF is the most complicated of the output formats supported by the p2p program. It is also the most extensible, since APF supports other sizes besides 320 x 200 and 640 x 200, and supports mode320, mode640, and mode3200 in a single file format. Scanlines are run-length encoded and no non-portable baggage like QuickDraw routines need to be interpreted. It also allows for user-defined chunks called BLOCKS.

```
/* APF structures */
/* FileType - $C0 AuxType $0002 */
/* integers are in big endian (Motorola) Format */

typedef struct tagPNTPIC
{
    ulong Length; /* Block Length */
    uchar Kind[5]; /* 4 'M' 'A' 'I' 'N' */
    ushort MasterMode; /* 0 for mode320 and 0x80 for mode640 */
    ushort PixelsPerScanline; /* 320 or 640 */
    ushort NumColorTables; /* 16 */
    uchar ColorTable[16][32]; /* $0RGB table buffer */
    ushort NumScanLines; /* vertical resolution */
    ushort ScanLineDirectory[200][2]; /* scbs */
} PNTPIC;

typedef struct tagPNTBROOKS
{
    ulong Length;
    uchar Kind[5]; /* 4 'M' 'A' 'I' 'N' */
    ushort MasterMode; /* 0 */
    ushort PixelsPerScanline; /* 320 */
    ushort NumColorTables; /* 1 */
    uchar ColorTable[1][32]; /* $0RGB table buffer */
    ushort NumScanLines; /* vertical resolution */
    ushort ScanLineDirectory[200][2]; /* sequential scbs */
} PNTBROOKS;

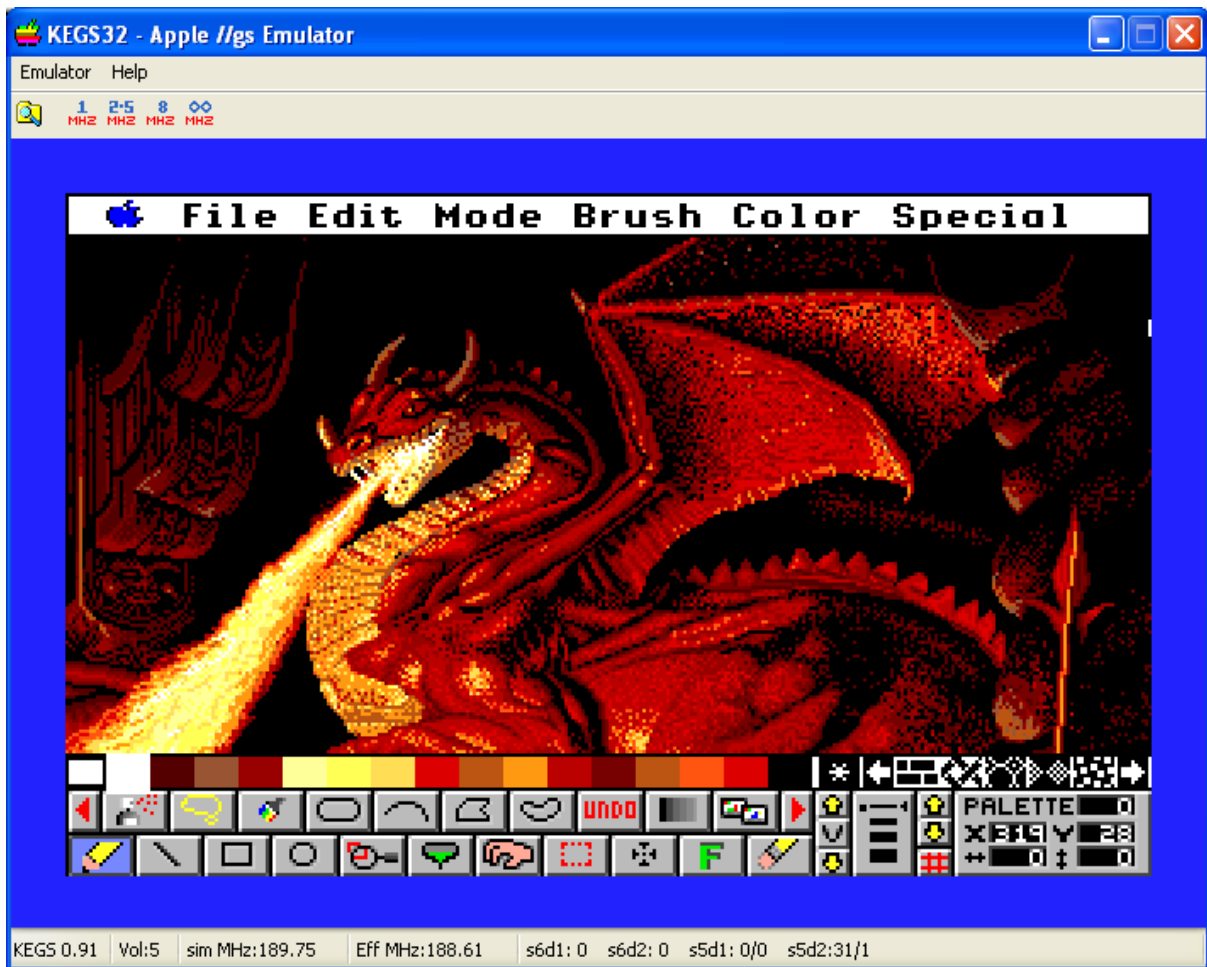
typedef struct tagMULTIPAL
{
    ulong Length;
    uchar Kind[9]; /* 8 'M' 'U' 'L' 'T' 'I' 'P' 'A' 'L' */
    ushort NumColorTables; /* 200 */
    uchar ColorTableArray[200][32]; /* $0RGB table buffer */
} MULTIPAL;
```

The APF consists of CHUNKS called BLOCKS. The MAIN Block in an APF contains an entire mode320 or mode640 PIC file equivalent and in the case of a mode3200 Brooks file equivalent, everything except for the 200 palettes is stored in MAIN.

An APF file can be used to store Image Fragments (Sprites) or screen-size and larger SHR images. All the SHR scanlines stored in an APF are in PackBytes RLE format, but the header information is not. However it uses Motorola Big-Endian which undoubtedly would be better understood by more Windows Users than Mac Users today

While APF's PackBytes RLE is not as efficient as DreamGraphx LZW compression, APF files are widely supported, including by DreamGraphx. (Proprietary DreamGraphx files are not supported by the p2p program.)

APF's store the position for the start of each scanline so that each scanline can be unpacked separately. This allows an APF loader to support scrolling through an SHR image larger than the screen. Memory is scarce on the Apple II, but because the APF stores the start of each scanline, an APF loader can avoid the need for much memory by seeking through a file to the start of a packed line, then reading and unpacking part of a line directly to the SHR screen.



## **Input/Output File Extension – PA3#C00004 – Brooks Eagle/PackBytes**

### Apple II File Type Notes

---

Technically Unsupported

File Type: \$C0 (192)  
Auxiliary Type: \$0004

Full Name: Packed Apple IIgs Super Hi-Res 3200 Color Screen Image File

Short Name: Packed Super Hi-Res 3200 color image

Written by: Bill Buckels

May 1, 2014

Files of this type and auxiliary type contain a packed Apple IIgs Super Hi-Res 3200 color screen image.

Files of type \$C0 and auxiliary type \$0004 contain a packed Apple IIgs Super Hi-Res 3200 color screen image, which is created by passing the entire file of type \$C1 and auxiliary type \$0002 through the PackBytes routine.

If you restore a file of this type to its original size of 38400 bytes with UnPackBytes, you can save the unpacked data to a file of type \$C1 and auxiliary type \$0002 (Apple IIgs Super Hi-Res 3200 color screen image).

Files of type \$C1 and auxiliary type \$0002 are often referred to as "Brooks format" after the designer of the format, John Brooks. The file structure is 32000 bytes of pixelData followed by 200 Color Tables. Each color table is stored in reverse order; the color value for color 15 is stored first.

The format for these files is similar to that for Super Hi-Res screen images except that there are no Scan Line Control Bytes ("scb's") and these have 200 Color Tables instead of 16. "Brooks" files and their Color Tables need to be constantly loaded directly the screen using a programming technique that is not easy; the programmer must constantly synchronize Color Table exchange with the Apple IIgs Super Hi-Res display routine intervals throughout the entire display duration, making these impractical for display except in programs that are written efficiently and capable of sharing precise display intervals with other functions.

### **Further Reference**

---

- Apple II File Type Notes, File Type \$C1, Auxiliary Type \$0002

## Additional SHR File Format Notes



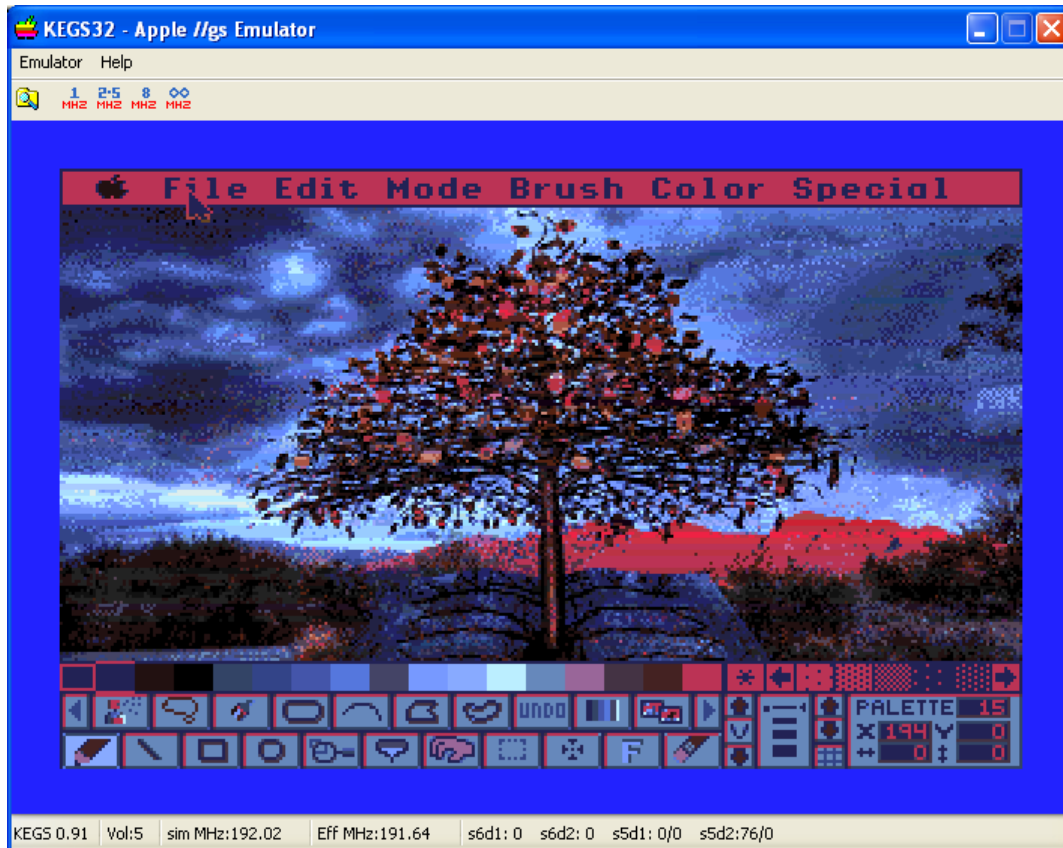
The two variants of the SHR QuickDraw PICT file format are not supported. QuickDraw II PICT files store pictures as a script; a series of command instructions (opcodes) which are used to record the QuickDraw II commands that created the picture. Modeled after PICT2 on the Macintosh, they can be used to exchange pictures between the IIGs and the Mac. These are loaded by “playing-back” in a IIGs GUI program with the QuickDraw II Toolbox Routine DrawPicture, but they are not bitmapped graphics files and they are not portable by any definition that I would care to implement or support.

The Activision Paintworks Packed Super Hi-Res Picture File is not supported. This file is targeted to the phantom users of Activision's PaintWorks Program (there likely aren't any), and contains header baggage reliant on QuickDraw in the form of QuickDraw II Patterns in addition to Bitmapped Graphics Information so it is not portable. It is also not as robust as the APF when it comes to sizes; only screen width is supported. Paintworks Packed files can support full-paged documents but do not specify exactly how many scanlines are on a page. Since the PaintWorks program itself does not provide proper support for standard SHR graphics formats or support more than 16 colors as near as I can tell, it is useless as a Paint Program, so there is no point in supporting PaintWorks.

The Apple II FileType notes can be referred to for additional details not included here or elsewhere in this program's source code comments or additional documentation.



## P2P – Pic to Pnt Portable HackBytes Example Program



P2P(C) Version 1.0 Copyright Bill Buckels 2014  
All Rights Reserved.  
Portable HackBytes(C) Copyright Bill Buckels 2014  
All Rights Reserved.  
UnPackBytes(c) Copyright (c) 2007, FaddenSoft, LLC  
All rights reserved.  
Raw format to Portable HackBytes Encoded format:  
Default:  
Usage: "p2p file.shr" or "p2p file.sh3"  
Output: file.pnt  
Option -p (Packed Screen)  
"p2p file.shr -p" or "p2p file.sh3 -p"  
Output: file.pak or file.pa3  
Option -u (UnPack)  
PackBytes Encoded format to Raw format:  
Usage: "p2p file.pnt -u" or "p2p file.pak -u"  
Output: file.shr  
"p2p file.pnt -u" or "p2p file.pa3 -u"  
Output: file.sh3  
Option -t (CiderPress File Attribute Preservation Tags)  
Option -s (-s1 to -s4 Singleton Thresholds)  
Option -q disable HackQuads (for demo purposes only)  
See Source Code and Documentation for More Information!

## Program Organization

Generally speaking p2p follows the program organization of any command-line conversion utility written in the C programming language. Like most command line utilities, p2p parses its inputs, outputs, and options before getting started.

After opening the input file in main(), p2p calls the ReformatSHR() function to read the SHR input file. If the input file is a compressed file, UnPackBytes() is called to unpack the file into a buffer. ReformatSHR() then writes the equivalent “raw” SHR output file; either a PIC or a “Brooks” file.

But if the input file is a “raw” SHR file, based on file size alone, ReformatSHR() decides whether the input file is a raw PIC file or a raw “Brooks” file and the appropriate proxy function is called which then writes the compressed output file with Portable HackBytes compression. The Proxy Functions that call HackBytes() are:

- **PackApf();**
- **PackPic();**
- **PackBrooks();**

This document does not aim to teach you how to write C programs. The implementation of common functionality like reading files and allocating memory is well understood. There are, however, some important details to do with the SHR file formats that can't necessarily be understood easily, so in this document, I am providing enough example code from p2p to explain how all of this works.

The p2p source code should also be reviewed on its own for additional information.

A final word on the p2p program's organization is “good manners”. Despite the fact that it is a demo program and lives in the wild, p2p is fairly well behaved.

Memory allocation is dynamic in most cases and reasonably modest, but would likely need to be refactored for use in a low memory environment. File I/O is extensive. Both are checked for common errors and if p2p can't safely run it likely won't work at all. If a file output error occurs the offending file will be closed and removed. Error messages are provided so a reasonable amount of diagnostics can be done if common problems are encountered.

P2p can also be easily modified to suit other nefarious purposes, and comes with many comment lines and a DEBUG mode for those who are inclined to re-use portable code. A single purpose or portable and redistributable SHR loader or editor can be relatively easily written by stripping-out useless lines and adding display and editing routines.

## The HackBytes() Function

This PackBytes encoder is really a simple list builder followed by a list processor. Although the list processor for this function was written for PackBytes Run Length Encoding (RLE), the list builder is a gutted version of the PCX encoder from the Z-Soft technical reference for the PCX file format.

```
/* The Crux of the Biscuit */
int HackBytes(uchar *inbuff, ushort inlen, ushort SingletonThreshold)
{
    uchar this,last,msk;
    ushort runcount, repeats, singlerun, maxpack;
    ushort idx,jdx,i;
    unsigned RawCount=0,SingleCount=0,PackedCount=0,QuadCount=0;

    /* ***** Build the List for this line ***** */
    /* ***** Build a list of count,value pairs ***** */

    RawCount = 0;
    last = inbuff[0];
    runcount=1;
    for(idx=1;idx<inlen;idx++){
        this=inbuff[idx];
        if(this==last){
            runcount++;
        }
        else{
            if(runcount > 0){
                RawBuf[RawCount].CNT = runcount;
                RawBuf[RawCount].VAL = last;
                RawCount++;
            }
            last=this;
            runcount=1;
        }
    }

    /* stragglers */
    if(runcount > 0) {
        RawBuf[RawCount].CNT = runcount;
        RawBuf[RawCount].VAL = last;
        RawCount++;
    }
}
```

After initially reading a raw Super Hi-Res scanline into a sequential list of Count, Value pairs, we have 2 types of nodes; single bytes and repeated bytes and we are now ready to process the list and build a packed scanline.

The idea here is to accumulate Singletons in a separate queue, until we hit a repeat, then flush the Singleton queue by encoding the Singletons first, followed by the repeat. At the end of the list, if we have any Singletons left in the Singleton queue, we finally flush the queue by encoding the remaining Stragglers

A Singleton can be a single group of bytes... a group of 1 to 4 bytes, although 2 bytes seems to be optimal so the default setting is 2 bytes. The setting for this is called "SingletonThreshold". In the p2p program this is a command line option so you can experiment with this and come to your own conclusion.

```

/* ***** */
/* ===== Encode the List for this line ===== */
/* ***** */

/* Process a list of count,value pairs */

SingletonThreshold++;
if (SingletonThreshold < 2 || SingletonThreshold > 5) {
    /* default - 1 and 2 bytes are encoded as Singletons */
    SingletonThreshold = 3;
}

PackedCount = SingleCount = 0;

for (idx=0;idx<RawCount;) {
    runcount = RawBuf[idx].CNT;
    if (runcount == 0) {
        /* list nodes should never have a zero count */
        idx++;
        continue;
    }

    if (runcount < SingletonThreshold) {
        /* push singleton nodes onto the stack
           until we hit a repeat node */
        for (i=0;i<runcount;i++) {
            RawBuf[SingleCount].Singleton = RawBuf[idx].VAL;
            SingleCount++;
        }
        idx++;
        continue;
    }
}

```

During encoding of Singletons, two different methods of encoding are tested, and the encoded Singletons are appended to the Packed Buffer using the most optimal:

Encoding	Description	Mask - Priority
00xxxxxx: (0-63)	1 to 64 bytes follow, all different	0x00 - 4
10xxxxxx: (0-63)	1 to 64 repeats of next 4 bytes (quad runs)	0x80 - 3

```

/* if we have hit a repeat list node... */
/* before encoding the repeat, pop singleton nodes (if any) off the
stack and encode 'em first */
/* two modes of encoding... Mask 0x80 and Mask 0x00 - decide which is
more efficient */
    while (SingleCount > 0) {

        /* ***** */
        /* ===== Singleton Option 1 - Mask 0x80 === */
        /* ***** */
        /* ===== Build a Quad Run of Singletons === */
        /* ***** */

        singlerun = 0; /* needed later on */

        /* check for repeats of 4 byte patterns in Singletons */
        QuadCount = HackQuads(SingleCount);

        if (QuadCount == 0) break;

        /* the following calculates the raw encoding for a run of singletons */
        /* we don't need to actually run singleton option 2 to get a line
length for comparison because that's pretty well defined */
        maxpack = (ushort)(SingleCount/64);
        if ((SingleCount % 64) != 0) maxpack++;
        maxpack+=SingleCount;

        /* if no efficiency gain, just encode as raw singletons */
        /* it could very well be that at the end of encoding the line that the
entire line gets replaced with a singleton run depending on how
expanded the encoded line becomes */

        if (QuadCount > maxpack) break;
        /* otherwise append encoded 4 byte patterns to the packed line */
        memcpy(&PackedBuf[PackedCount], &PackedBuf4[0], QuadCount);

        /* Advance the count and pop the Singleton Stack */
        PackedCount += QuadCount;
        SingleCount = 0;
        break;
    }

```

In the code above you can see that the HackQuads() helper function is called to build a temporary packed buffer of Singletons primarily using Quad runs (repeats of 4 different bytes). Since there is no simple formula for the efficiency of Quad runs, we must run the Quad run encoder for comparison so while we are at it, we build the temporary buffer each time through. If encoding of Singletons using Quad runs is more efficient than encoding of Singletons as a chunk of encoded single bytes then the temporary buffer is appended to the packed buffer and the Singleton Queue is flushed before encoding the impending Repeat Node. Otherwise, if no efficiency can be gained using Quad runs, a chunk of single bytes is encoded:

```

while (SingleCount > 0) {

    /* ***** */
    /* ===== Singleton Option 2 - Mask 0x00 ===== */
    /* ***** */
    /* ===== Build a Raw Run of Singletons ===== */
    /* ***** */

    if (SingleCount < 65) {
        msk = (uchar)(SingleCount - 1);
        PackedBuf[PackedCount] = msk;
        PackedCount++;
        for (i=0;i<SingleCount;i++) {
            PackedBuf[PackedCount] = RawBuf[singlerun].Singleton;
            singlerun++;
            PackedCount++;
        }
        SingleCount = 0;
        break;
    }

    PackedBuf[PackedCount] = (uchar)63;
    PackedCount++;
    for (i=0;i<64;i++) {
        PackedBuf[PackedCount] = RawBuf[singlerun].Singleton;
        singlerun++;
        PackedCount++;
    }
    SingleCount -= 64;
}
}

```

Now that the Singleton nodes have been encoded and the Singleton Queue is empty we follow a formula to encode the repeats. This is quite a complicated formula more easily explained in code but requires some knowledge of the efficiency of PackBytes encoding to understand.

Encoding	Description	Mask – Priority
01xxxxxx: (0-63)	1 to 64 repeats of next byte	0x40 - 2
11xxxxxx: (0-63)	1 to 64 repeats of next byte taken as 4 bytes	0xc0 - 1

For runs of 256 or more repeated bytes the most efficient option is a second type of Quad only 2 bytes long called a Quad Count Repeat. Since a normal Single Count Repeat can only encode up to 64 repeated bytes into 2 bytes, Quad Count Repeats are used in HackBytes() whenever it is possible to gain efficiency.

For runs of 2 to 64 repeated bytes, a normal Single Count Repeat is 100% efficient.

For runs of 65 to 255 repeated bytes, either 2 or 4 encoded bytes are 100% efficient. Quad Count Repeats are used to encode the portion of the repeat that is equally divisible by 4, and Stragglers of 2 and 3 bytes, if any, encode in 2 bytes. Stragglers of 1 byte are left raw and pushed into the Singleton Queue and wait for more Singletons.

```

/* Hi-Low Split */

/* ***** */
/* ===== Quad Count Repeats Mask 0xc0 ===== */
/* ***** */
/* ===== Build Full Runs of Repeated Pairs */
/* ***** */

/* Mask 0xc0 - use full quads to reduce repeats */
while (runcount > 256) {
    PackedBuf[PackedCount] = 0xff; /* 63 | 0xc0 */
    PackedCount++;
    PackedBuf[PackedCount] = RawBuf[idx].VAL;
    PackedCount++;
    runcount -= 256; /* decrement runcount until 256 or below */
}
/* 1 byte runs are a loss at the end of any repeat run...
PUSH 1 BYTE onto the singleton stack and give it a second chance */
if (runcount < 2) {
    if (runcount == 1) {
        /* push singletons on the stack */
        RawBuf[SingleCount].Singleton = RawBuf[idx].VAL;
        SingleCount++;
    }
    idx++;
    continue;
}

/* ***** */
/* ===== Single Count Repeats Mask 0x40 ===== */
/* ***** */
/* ===== Build Low Runs of Repeated Pairs */
/* ***** */

/* Mask 0x40 for repeats of 2 to 64 */
if (runcount < 65) {
    msk = (uchar)(runcount - 1);
    PackedBuf[PackedCount] = (uchar)(msk | 0x40);
    PackedCount++;
    PackedBuf[PackedCount] = RawBuf[idx].VAL;
    PackedCount++;
    idx++;
    continue;
}

/* End of High-Low Split */

/* ***** */
/* ===== Quad Count Repeats Mask 0xc0 ===== */
/* ***** */
/* ===== Build Low Runs of Quad Pairs ===== */
/* ***** */

/* Mask 0xc0 - use quads for repeats of 65 to 255 */
repeats = runcount / 4;

```

```

msk = (uchar) (repeats - 1);
PackedBuf[PackedCount] = (uchar) (msk | 0xc0);
PackedCount++;
PackedBuf[PackedCount] = RawBuf[idx].VAL;
PackedCount++;
runcount -= (repeats * 4);

/* a 1 byte run is a loss */
if (runcount < 2) {
    if (runcount == 1) {
        /* push stragglers on the stack */
        RawBuf[Singleton].Singleton = RawBuf[idx].VAL;
        Singleton++;
    }
    idx++;
    continue;
}

/* ***** */
/* ===== Straggler Repeats Mask 0x40 ===== */
/* ***** */
/* ===== Build Quad Overflow Trailer ===== */
/* ***** */

/* Mask 0x40 for stragglers - repeats of 2 or 3 */
/* this breaks even or gains a byte in efficiency */
msk = (uchar) (runcount - 1);
PackedBuf[PackedCount] = (uchar) (msk | 0x40);
PackedCount++;
PackedBuf[PackedCount] = RawBuf[idx].VAL;
PackedCount++;
/* on to the next list member */
idx++;
}

```

The line is now entirely packed except for Singletons if any, so these Stragglers need to be appended to the packed line before returning the packed line and the packed length to the caller. The following code duplicates the code in the main processing loop above; first Singleton Quads are tried, and compared to Singleton Chunk encoding and the most efficient segment is appended to the packed line:

```

/* clear stragglers */

/* two modes of encoding... Mask 0x80 and Mask 0x00 -
   decide which is more efficient */
while (Singleton > 0) {

    /* ***** */
    /* ===== Build a Quad Run of Stragglers ===== */
    /* ***** */
    singlerun = 0; /* needed later on */

    /* check for repeats of 4 byte patterns in Singletons */
    QuadCount = HackQuads(Singleton);
}

```



```

        if (QuadCount == 0) break;

/* the following calculates the encoding for a run of singletons */
maxpack = (ushort)(SingleCount/64);
if ((SingleCount % 64) != 0) maxpack++;
maxpack+=SingleCount;

/* if no efficiency gain, just encode as singletons */
if (QuadCount > maxpack) break;

/* otherwise append encoded 4 byte patterns to the packed line */
memcpy(&PackedBuf[PackedCount], &PackedBuf4[0], QuadCount);

/* Advance the count and pop the Singleton Stack */
PackedCount += QuadCount;
SingleCount = 0;
break;
}

while (SingleCount > 0) {

    /* ***** */
    /* ===== Build a Raw Run of Straggletons */
    /* ***** */

/* pop any remaining singleton nodes off the stack and finish-up */
    if (SingleCount < 65) {
        msk = (uchar)(SingleCount - 1);
        PackedBuf[PackedCount] = msk;
        PackedCount++;
        for (i=0;i<SingleCount;i++) {
            PackedBuf[PackedCount] = RawBuf[singlerun].Singleton;
            singlerun++;
            PackedCount++;
        }
        SingleCount = 0;
        break;
    }

    PackedBuf[PackedCount] = (uchar)63;
    PackedCount++;
    for (i=0;i<64;i++) {
        PackedBuf[PackedCount] = RawBuf[singlerun].Singleton;
        singlerun++;
        PackedCount++;
    }
    SingleCount -= 64;
}
}

```

Now that we are done encoding the line, we know how many bytes it packed to. If the encoded line expanded beyond the length of a raw chunk of encoded Singletons, we just encode it as a Raw Chunk of Singletons. For a production version we would likely also want to test for an efficiency gain using Singleton Quads alone under some circumstances, especially on shorter lines. I could get into a protracted discussion about when that would be better and when that would be much worse, but in the interests of

keeping this simple, the Raw Chunk of Singletons is the last optimization I have implemented for this version.

Additional optimizations are also performed within the p2p program itself which will be discussed briefly later on.

```
/* ***** */
/* ===== Build a Raw Line of Singletons ===== */
/* ***** */

/* a raw line of singletons incurs 1 count byte for every 64 data bytes
*/
/* a line of 160 singletons is expanded by 3 count bytes to 163 bytes */
/* if the packed line expands beyond this length then simply encode it
as singletons */
/* the following calculates the most efficient encoding for a line of
singletons */
    maxpack = (ushort)(inlen/64);
    if ((inlen % 64) != 0) maxpack++;
    maxpack+=inlen;
/* if the length of a packed line goes beyond this, run individual
bytes */
    if (PackedCount > maxpack) {

        PackedCount = RawCount = 0;
        maxpack = (ushort) (inlen / 64);
        for (idx = 0;idx < maxpack; idx++) {
            /* runs of 64 individual bytes */
            PackedBuf[PackedCount] = 63; PackedCount++;
            for (jdx = 0; jdx < 64; jdx++, RawCount++, PackedCount++) {
                PackedBuf[PackedCount] = inbuff[RawCount];
            }
        }
        inlen -= (maxpack*64);
        if (inlen > 0) {
            /* stragglers if any */
            PackedBuf[PackedCount] = (uchar)(inlen-1); PackedCount++;
            for (jdx = 0; jdx < inlen; jdx++, RawCount++, PackedCount++)
            {
                PackedBuf[PackedCount] = inbuff[RawCount];
            }
        }
    } /* end of raw chunk */

/* in either case return length of packed buffer */
return PackedCount;
}
```

## HackQuads and the Mid-Stream Encoder

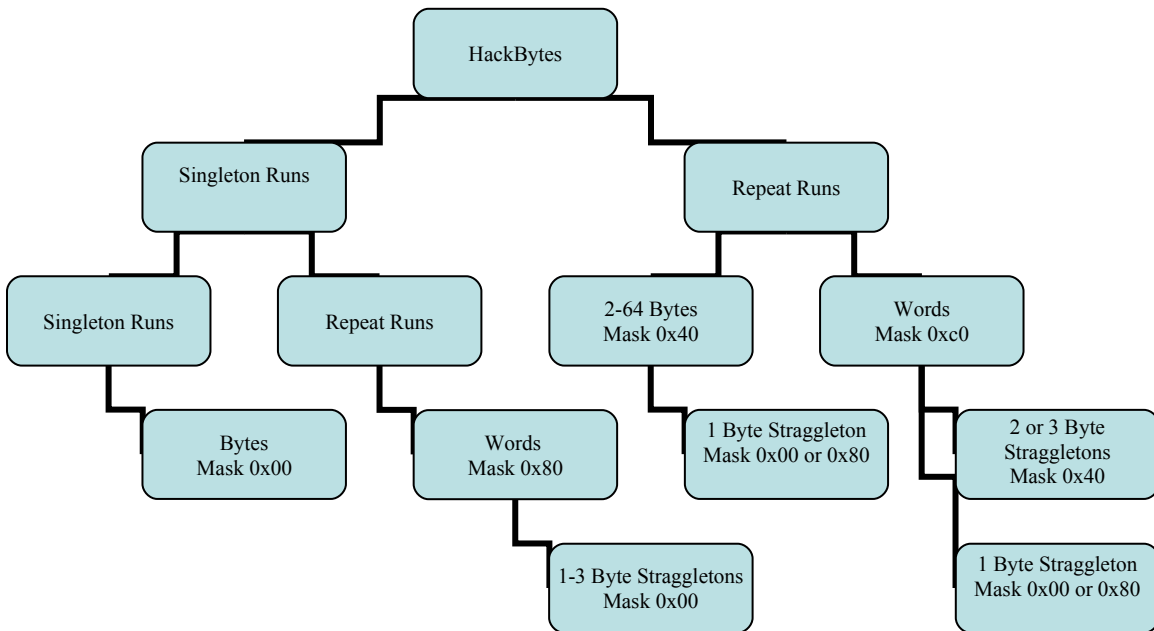
The HackQuads() Mid-Stream Encoder encodes repeated runs of 4 byte Words (Quad Singletons):

Encoding	Singletons	Mask
10xxxxxx: (0-63)	1 to 64 repeats of next 4 bytes (quad runs)	0x80

“Classic” run length encoding schemes are somewhat simpler than PackBytes encoding but the idea is still the same; encode Singletons and Repeats separately.

I really hope you haven’t been reading this document up until now in horror and bewilderment, already abandoning all hope thinking black magic is afoot here! The only thing afoot here is common sense.

Anything you read on PackBytes would probably lead you to believe that 4 different StereoTypes of mutually exclusive encoding exist. If you do think in StereoTypes and sets and aggregates, common sense says HackBytes actually has only two types of encoding (not 4):



## HackBytes and the Main Stream Encoder



Encoding	Singletons	Mask
00xxxxxx: (0-63)	1 to 64 bytes follow, all different	0x00
	<b>Repeats</b>	
01xxxxxx: (0-63)	1 to 64 repeats of next byte	0x40
11xxxxxx: (0-63)	1 to 64 repeats of next byte taken as 4 bytes	0xc0

During the initial development I organized HackBytes() like a “Classic” encoder by using a list processor and only 3 of the 4 PackBytes RLE encoding schemes. These 3 encoding schemes (above) functioned well enough together and included all but two of the current optimizations. The 2 missing optimizations were:

1. SingletonThreshold
2. HackQuads() Mid-Stream Encoder

Without the 2 missing optimizations Portable HackBytes() was less efficient than Apple Computer’s PackBytes. The 19 files that were used for my initial comparison test are the same 19 files that are used in my final comparison test shown at the end of this document.

After adding these 2 missing optimizations, Portable HackBytes() was more efficient than Apple Computer’s PackBytes.

You can turn-off each of the 2 missing optimizations to compare the difference for yourself by using the p2p program’s command options `-s1` and `-q`.

All of the optimizations used in Portable HackBytes() and the p2p demo program appear throughout this document and are also listed in the Optimizations Table shown below.

## **Portable HackBytes() Optimizations Table**

The following is a table of all the optimizations in Portable HackBytes and the p2p program:

<b>Optimizations</b>	<b>Description</b>
Separate Repeats from Singletons	Pre-Process Raw Chunk into sequential list separating Repeats from Single Bytes Leave Raw Buffer intact for Raw Run Optimization
Encode from List	Process the sequential list instead of using the raw chunk Save the raw chunk for Raw Run Comparison Provide a sequential Singleton Stack for single runs Provide a temporary encoded buffer for HackQuads()
SingletonThreshold	Allow Singleton Stack to store runs of 1 to 4 bytes Use the optimal default of 2 bytes Allow a Command Line over-ride of 1 to 4 bytes Do not use this setting for Straggletons in Repeats
Singletons	Push Singletons onto Singleton Stack until a Repeat Pop Singleton Stack and Encode before Encoding Repeat Pop Straggletons and Encode at the end of the run
HackQuads() Mid-Stream Encoder	Compare Encoding runs of Single Bytes to Quad runs Encode Quad run first and precalculate Single run. If Quad run is smaller use instead of Single run Allow this option to be disabled from the Command Line
Repeats (hi-lo split)	Encode Repeats Greater than 256 using Mask 0xc0 Encode Repeats Greater than 64 using Mask 0xc0 Encode Repeats of 2 to 64 using Mask 0x40 Encode Straggletons of 2 and 3 using Mask 0x40 Push Straggletons of 1 onto Singleton Stack
Raw Run	Compare efficiency of encoded line to worst case Singleton Use raw run encoded as Singletons if it is smaller
Separate Scan Lines From ColorTables and From scbs	Encode ColorTables and ScanLine Control bytes as Separate Chunks for Maximum Repeat Alignment for Eagle/PackBytes File Output
Eagle/PackBytes Dry Run Encoding 4 – line encoded chunks	Encode and Compare Efficiency 1,2 or 4 line segments + scb + palette Write most efficient to file.
Eagle/PackBytes Wet Run Encoding File size encoded chunk	Encode and Compare Efficiency against 4 – line chunks Dry Run Encoding of Entire File If more efficient, overwrite 4 – line chunk encoded file

Two of the above optimizations, Dry-Run Encoding and Wet-Run Encoding, apply only to Eagle/PackBytes PAK and PA3 file output. Because APF files only encode individual scanlines, chunks larger than a scanline cannot be used, so this optimization does not apply to them.

“Raw Run” optimization was initially targeted at APF files of 160 byte scanlines which have a maximum encoded raw length of 163 bytes. Some other analysis I have seen on PackBytes also refers to this efficiency of expanded scanlines. “Raw Run” optimization works best with shorter runs. When used with longer runs it can’t take advantage of repeats within the run, so Dry-Run encoding based on line-length segments will likely benefit, and the HackQuads() optimization for mid-stream encoding of Singletons is a short run variation of “Raw Run”, but if it works with Wet-Run encoding, the encoded file was better left as a raw unpacked file.

Dry-Run encoding works better with scan-lines than it does with 256 byte sectors or 512 byte blocks because the 2 dimensional geometry of an image generally aligns on scan-line boundaries. To make this more efficient for long repeats, since 256 byte runs are optimal for Quad Count Repeats in the PackBytes RLE, I have segmented this optimization into 50 segments which are equally divisible by 128 but optimally divisible by 256 using a factor of 2.5 which produces an error of 50% and a threshold of 20%. This was “fine-tuned” from a 1280 byte segment of 8 lines which produced larger encoded results than a 640 byte segment of 4 lines. There seems to be an optimal frequency for segments of 4 lines. Beyond that, encoding the entire scanline area of the file seems to work better than encoding 8 line segments.

The “Singleton Threshold” optimization with a default of 2 bytes works on its own, and works even better with the HackQuads() Mid-Stream encoder optimization. Repeated 4 byte runs can hide in Singletons and byte pairs. For greater efficiency, byte pairs needed to be included in Singletons and a Mid-Stream Encoder was needed to sort through Singletons and look-ahead for Quad runs. When Quad runs are found Mid-Stream, the encoded runs for that segment are compared against an encoded “Raw Run” of only part of the chunk to see which method produces a more efficient smaller Packed Run. Using a “Singleton Threshold” of 1 byte, or 3 or 4 bytes creates larger results than 2 bytes. Using HackQuads() optimization creates smaller results than with HackQuads() disabled.

One of the reasons that “Singleton Threshold” works best with 2 bytes is how repeated runs of “pure” color tend to fall together with 4 byte repeats like in a dithered image. A run of pure color seems to generally be over 2 bytes. A dither seems to be 2 bytes or even 1 byte, but trailing repeats of 2 bytes do not seem to encode as efficiently as trailing repeats of 1 byte so “Singleton Threshold” is not applied to Straggletons.

*My observations are of course by no means speculative, which is precisely why I have provided a relatively complete HackBytes implementation in the p2p program so you can try the darned thing yourself. It was certainly easier to provide something usable and expandable, than to waste everyone’s time collating the seemingly endless data sets and probability theories that led to this, but that nobody wants to read anyway, and now lay abandoned on my hard-drive along with about 30 years of other caka. As for my optimizations; like everything else in HackBytes, they are just common sense, and better.*

## The HackQuads() Function



The HackQuads() function builds a run of Singletons into a temporary Quad Encoded line section as a candidate for efficiency comparison. If the Quad encoded line is more efficient than the equivalent “raw” encoded line it is used instead. The HackQuads() function called by HackBytes() is merely a wrapper for the PackQuads() Mid-Stream Encoder. You should recognize this encoder... it works the same way as the list builder for HackBytes() except that it compares 4 bytes at a time for repeats instead of one:

```
/* Helper Function called By HackQuads() */
unsigned PackQuads(uchar *inbuff, unsigned NumQuads, uchar *outbuff)
{
    unsigned runcount, x, idx, PackedCount = 0;
    uchar msk, this[4], last[4];

    /* ***** */
    /* ===== RLE for Quad Runs in Singletons == */
    /* ***** */

    memcpy((uchar *)&last[0], (uchar*)&inbuff[0], 4);
    runcount = 1;

    /* When I tried shifting the line using a more complicated
    segmented algorithm, the savings were at best only a few bytes on some
    images and none at all on others. It really wasn't worth the extra
    overhead, so I decided to use classic run-length encoding below for the
    sake of readability. */
```

```

for (x = 1, idx = 4; x < NumQuads; x++, idx+=4) {
    memcpy((uchar *)&this[0], (uchar *)&inbuff[idx], 4);
    if (memcmp(&this[0], &last[0], 4) == 0) {
        runcount++;
        if (runcount == 64) {
            msk = (uchar) (runcount - 1);
            outbuff[PackedCount] = (uchar) (msk | 0x80);
            PackedCount++;
            memcpy((uchar*)&outbuff[PackedCount], (uchar *)&this[0], 4);
            PackedCount+=4;
            runcount = 0;
        }
    }
    else {
        if (runcount > 0) {
            msk = (uchar) (runcount - 1);
            outbuff[PackedCount] = (uchar) (msk | 0x80);
            PackedCount++;
            memcpy((uchar *)&outbuff[PackedCount], (uchar *)&last[0], 4);
            PackedCount+=4;
        }
        memcpy((uchar *)&last[0], (uchar *)&this[0], 4);
        runcount = 1;
    }
}

/* straggler Quads */
/* straggler singletons are appended to the line
   after returning to HackQuads() */
if(runcount > 0) {
    msk = (uchar) (runcount - 1);
    outbuff[PackedCount] = (uchar) (msk | 0x80);
    PackedCount++;
    memcpy((uchar *)&outbuff[PackedCount], (uchar *)&this[0], 4);
    PackedCount+=4;
}

return PackedCount;
}

```

Here's the wrapper for the above encoder. It's not merely a wrapper. It encodes stragglers as single byte runs. Since this precedes a conventional repeat of a single byte, it must be de-queued now!

```

/* Helper Function called By HackBytes */
/* Singletons Only - Repeated Patterns of 4 Bytes - Mask 0x80 */
unsigned HackQuads(unsigned SingleCount)
{
    unsigned singlerun, NumQuads, remaining, runcount, PackedCount = 0;
    uchar msk;

    if (disable_HackQuads == 1) return 0; /* for demo purposes */
    /* if flag is not set, no memory is allocated so just return */
    if (Pack4Singletons == INVALID) return 0;

```



```

/* in order for a repeat of 4 to occur we need a minimum of 8 bytes */
NumQuads = SingleCount / 4;
if (NumQuads < 2) return 0;

/* expand bytes into look-ahead mini-buffer */
/* singlerun starts at base 0 */
for (singlerun = 0; singlerun < SingleCount; singlerun++) {
    RawBuf4[singlerun] = RawBuf[singlerun].Singleton;
}

/* RLE for Quad pattern runs */
singlerun = (NumQuads * 4);
remaining = SingleCount - singlerun;

/* shifted segment optimization here makes little difference */
/* see notes in PackQuads (above) */
/* however this is an interesting area of the code to play with */
/* I have left a "ping" test in place below left over from my last
test code before the production version of this function for possible
future drill-down to try some pattern matching stuff here... but I
thought I'd better keep it simple for the first release. */

PackedCount =
PackQuads((uchar *)&RawBuf4[0], NumQuads, (uchar *)&PackedBuf4[0]);

/* Straggletons - encode as a single run of 1-3 bytes */
if (remaining > 0) {
    msk = (uchar) (remaining - 1);
    PackedBuf4[PackedCount] = msk;
    PackedCount++;
    memcpy((uchar *)&PackedBuf4[PackedCount],
           (uchar *)&RawBuf4[singlerun], remaining);
    PackedCount+=remaining;
}

#ifdef DEBUG

/* if you are mucking about with PackQuads set DEBUG and
redirect to a file for clues */

if (UnPackBytes((uchar *)&UnPackBuf4[0],
                (uchar *)&PackedBuf4[0], (long)
                SingleCount, (long) PackedCount) != 0) {
    puts("PackBytes Quad Error!");
    return 0;
}

if (memcmp((uchar *)&RawBuf4[0],
           (uchar *)&UnPackBuf4[0], SingleCount) != 0) {
    puts("Compare Quad Error!");
    return 0;
}

#endif
return PackedCount;
}

```

## Encoding 256 Color PAK Files with “Dry-Runs” and “Wet-Runs”

Since the Eagle/PackBytes file format is a “blob” that is packed end to end using PackBytes, it dawned on me that unlike the APF format, I had no restrictions to encode on a line by line basis but I had no requirement to just blindly encode the entire file in a big chunk. So I did both, and compared the two, and provide the most efficient file:

```
int PackPic()
{
FILE *fp;
ushort len, linelen, offset, inset, x, y, singlelen, doublelen, seglen,
segments, packet;
int status = FAILURE;

if (PackAlloc(32768) == INVALID) return status;

fp = fopen(pntfile,"wb");
if (NULL == fp) {
PackFree();
printf("Unable to open %s for output!\nExiting...\n",pntfile);
return status;
}

status = SUCCESS;

/* encoding by 1,2 or 4 line segments + scb + palette */
linelen = 0;
for (y = 0, offset = 0; y<50; y++,offset+=640) {

/* 4 segments of 1 line */
for (x = 0, inset = offset, singlelen = 0; x < 4; x++, inset+=160) {
singlelen += HackBytes((uchar *)&shrbuf[inset],
(ushort)160,use_threshold);
}

/* 2 segments of 2 lines */
for (x = 0, inset = offset, doublelen = 0; x < 2; x++, inset+=320) {
doublelen += HackBytes((uchar *)&shrbuf[inset],
(ushort)320,use_threshold);
}

/* 1 segment of 4 lines */
seglen = len = HackBytes((uchar *)&shrbuf[offset],640,use_threshold);
segments = 1;

if (doublelen < seglen) {
seglen = doublelen;
segments = 2;
}
if (singlelen < seglen) segments = 4;

packet = (ushort) 640 / segments;
```

```

for (x = 0, seglen = 0, inset = offset; x < segments; x++,
inset+=packet) {

    if (segments > 1) {
        len = HackBytes((uchar *)&shrbuf[inset],packet,use_threshold);
    }
    seglen += len;
    fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
    if (ferror(fp)) {
        status = INVALID;
        break;
    }
}
if (status == INVALID) break;

linelen += seglen;
}

```

The code above segments 4 scanlines of an SHR image into a segment for encoding. It compares the efficiency of encoding each scanline separately, and in pairs, with the efficiency of encoding all 4 scanlines together. The most efficient method is then used to encode the 4 line segment. By using scanline boundaries the two dimensional image geometry is aligned which generally yields better compression results than using sector or block boundaries. The 640 byte segment size is a compromise between the sector alignment efficiency of PackBytes and the geometric screen width efficiency of 160 bytes for graphics object and primitives, and 2 dimensional image alignment.

Again, this can be optimized much further by using a larger segment and many more comparisons, but since this already yields better results than Apple Computer's PackBytes and is still quite readable, I have done no further optimization for this release.

```

if (status == SUCCESS) {
for (offset = 32000;;) {
    len = HackBytes((char *)&shrbuf[offset],256,use_threshold);
    fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
    if (ferror(fp)) {
        status = INVALID; break;
    }
    linelen += len;
    offset +=256;
    len = HackBytes((uchar *)&shrbuf[offset],512,use_threshold);
    fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
    if (ferror(fp)) {
        status = INVALID;
        break;
    }
    linelen+=len;
    break;
}
}

fclose(fp);

```

The code above uses the boundaries of the scb's and the ColorTables in an SHR image to attempt an efficiency gain. Unused palettes in the ColorTable will compress well, and scb's that all use the same palette will also compress well. Otherwise, the only potential for efficiency gain is the 56 bytes of padding at the end of the scb block, and whatever other random repeats may occur.

The code above has compressed an SHR file to an Eagle/PackBytes format file using segmented optimization. But more efficiency may be gained by simply encoding the entire file as a single PackBytes "blob":

```
while (status == SUCCESS) {
printf("RLE by line segments = %u\n",linelen);

/* if encoding the entire file into a single blob results in a
   smaller size than segmented encoding by scanline boundaries
   then overwrite the segmented file with with an encoded blob */

len = HackBytes((uchar *)&shrbuf[0],(ushort)32768,use_threshold);
printf("RLE by file = %u\n",len);

if (len < linelen) {
    fp = fopen(pntfile,"wb");
    /* if we can't open the file at this stage of the game
       then just leave it alone... must be an access problem */
    if (NULL == fp) {
        puts("Segmented Encoding Used!");
    }
    else {
        fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
        if (ferror(fp)) status = INVALID;
        else puts("File Encoding Used!");
        fclose(fp);
    }
}
else {
    puts("Segmented Encoding Used!");
}
break;
}

if (status == SUCCESS) printf("Created: %s\n",pntfile);
else printf("Can't Create %s!\nOutput File Write Error!
Exiting...\n",pntfile);
PackFree();

return status;
}
```

### **Encoding 3200 Color Brooks PAK Files**

A Brooks file is encoded to Eagle/PackBytes using roughly the same technique as the 256 Color SHR file shown on the preceding code:

```

int PackBrooks()
{
FILE *fp;
ushort len, linelen, offset, inset, x, y, singlelen, doublelen, seglen,
segments, packet;
int status = FAILURE;

if (PackAlloc(32000) == INVALID) return status;

fp = fopen(pntfile,"wb");
if (NULL == fp) {
PackFree();
printf("Unable to open %s for output!\nExiting...\n",pntfile);
return status;
}

status = SUCCESS;

/* encoding by 1,2 or 4 line segments + scb + palette */
linelen = 0;
for (y = 0, offset = 0; y<50; y++,offset+=640) {
/* 4 segments of 1 line */
for (x = 0, inset = offset, singlelen = 0; x < 4; x++, inset+=160) {
singlelen +=
HackBytes((uchar *)&shrbuf[inset],(ushort)160,use_threshold);
}
/* 2 segments of 2 lines */
for (x = 0, inset = offset, doublelen = 0; x < 2; x++, inset+=320) {
doublelen += HackBytes((uchar *)&shrbuf[inset],
(ushort)320,use_threshold);
}
/* 1 segment of 4 lines */
seglen = len = HackBytes((uchar *)&shrbuf[offset],640,use_threshold);
segments = 1;
if (doublelen < seglen) {
seglen = doublelen;
segments = 2;
}
if (singlelen < seglen) segments = 4;
packet = (ushort) 640 / segments;
for (x = 0, seglen = 0, inset = offset; x < segments; x++,
inset+=packet) {
if (segments > 1) {
len = HackBytes((uchar *)&shrbuf[inset],packet,use_threshold);
}
seglen += len;
fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
if (ferror(fp)) {
status = INVALID; break;
}
}
}
if (status == INVALID) break;

linelen += seglen;
}

```

```

if (status == SUCCESS) {
/* this could probably be optimized somewhat more but at this point I
am not sure how much use anyone would have for a Brooks file in PAK
format so just getting this off my plate for now */
offset = len = HackBytes((char*)&shrbuf[32000],
(usshort)6400,use_threshold);
fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
if (ferror(fp)) status = INVALID;
else linelen += len;
}

fclose(fp);

while (status == SUCCESS) {
printf("RLE by line segments = %u\n",linelen);

/* if encoding the entire file into a single blob results in a
smaller size than segmented encoding by scanline boundaries
then overwrite the segmented file with with an encoded blob */

len = HackBytes((uchar *)&shrbuf[0],(usshort)32000,use_threshold);
offset += len;
printf("RLE by file = %u\n",offset);

if (offset < linelen) {
for (;;) {
fp = fopen(pntfile,"wb");
/* if we can't open the file at this stage of the game
then just leave it alone... must be an access problem */
if (NULL == fp) {
puts("Segmented Encoding Used!"); break;
}
fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
if (ferror(fp)) {
fclose(fp); status = INVALID; break;
}
len = HackBytes((char *)&shrbuf[32000],
(usshort)6400,use_threshold);
fwrite((uchar *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
if (ferror(fp)) {
fclose(fp); status = INVALID; break;
}
puts("File Encoding Used!"); fclose(fp); break;
}
}
else { puts("Segmented Encoding Used!"); }
break;
}

if (status == SUCCESS) printf("Created: %s\n",pntfile);
else printf("Can't Create %s!\nOutput File Write Error!
Exiting...\n",pntfile);
PackFree();

return status;
}

```

## **Encoding APF Files**

APF files are required to unpack on scan-line boundaries, and their headers must remain uncompressed. This results in an unacceptable compression loss when compared to a PAK file of an SHR screen and in many cases a “raw” SHR screen file. Like any other SHR file, you need to put some thought into what you will be using these for.

You are unlikely to be concerned with seriously using APF files as a file interchange format except for nostalgic reasons so creating an APF file in this day and age is just for fun. For slideshows a “raw” SHR file is easiest to deal with. And LZ4 has every other compression format beat for overall efficiency for Apple IIGs games programming.

The APF header requires significant interpretation which, when combined with the inherent potential for efficiency loss of restricting encoding to line boundaries, makes its use even more impractical unless for describing and storing images of a different size than the SHR screen or for applications which can’t deal with other SHR formats or for reasons of personal preference or insanity.

Unless you interpret the APF file header, you can’t tell whether the APF is a mode640 or a mode320 or a mode3200 APF file. For completely device dependent screen size SHR files you don’t care about this.

Since we are encoding rather than decoding that isn’t an issue here. We already know what we are converting based on our input file. But organizing a “raw” device dependent file like an SHR file into some contraption like the APF that compromises the efficiency of PackBytes compression with a large interpreted header violates the primary purpose of Run-Length encoding (RLE) which is to store efficiently and unpack quickly. All other RLE formats that I can think of (except for Eagle/PackBytes) have the same problem, including Windows BMP’s.

So in most cases the following encoder which produces an APF file from a “raw” SHR file is simply a demo with a recipe for something practical rather than anything useful.

```
int PackApf()
{
FILE *fp;
ulong blocklen = 0L;
ushort len, y;
uchar ch;
int status = INVALID;
if (PntAlloc() == INVALID) return FAILURE;
fp = fopen(pntfile,"wb+");
if (NULL == fp) { PntFree();
printf("Unable to open %s for output!\nExiting...\n",pntfile);
return FAILURE;
}
}
```

```

switch(input_format) {
case BROOKS_FMT: brooks = (BROOKSFILE *)&shrbuf[0];
    SetUpBrooksPalette();
    fwrite((char*)&brooksMain[0].Length,sizeof(PNTBROOKS),1,fp);
    if (ferror(fp)) break; blocklen = (ulong)sizeof(PNTBROOKS);
    for (y=0;y<200;y++) {
        len =
        HackBytes((uchar *)&brooks[0].line[y][0],160,use_threshold);
        fwrite((char *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
        if (ferror(fp)) { status = FAILURE; break; }
        brooksMain[0].ScanLineDirectory[y][0] = Motorola16(len);
        brooksMain[0].ScanLineDirectory[y][1] = Motorola16(y);
        blocklen += len;
    }
    if (status == FAILURE) { status = INVALID; break; }
    fwrite((char*)&MultiPal[0].Length,sizeof(MULTIPAL),1,fp);
    if (ferror(fp)) break; rewind(fp);
    brooksMain[0].Length = Motorola32(blocklen);
    fwrite(&brooksMain[0].Length,sizeof(PNTBROOKS),1,fp);
    if (ferror(fp)) break;
    status = SUCCESS; break;
case PIC_FMT:
default: pic = (PICFILE *)&shrbuf[0];
/* check first scb and use for MasterMode - as good as anything... */
ch = pic[0].scb[0];
if ((ch >> 7) > 0) { /* Set MasterMode to 0x80 if mode640 */
    picMain[0].MasterMode = Motorola16((ushort)0x80);
    /* Reset Horizontal Resolution to 640 pixels */
    picMain[0].PixelsPerScanline = Motorola16((ushort)640);
}
/* copy palette from picfile */
memcpy(&picMain[0].ColorTable[0][0],&pic[0].pal[0][0],512);
fwrite((char*)&picMain[0].Length,sizeof(PNTPIC),1,fp);
if (ferror(fp)) break;
blocklen = (ulong)sizeof(PNTPIC);
for (y=0;y<200;y++) {
    len = HackBytes((uchar *)&pic[0].line[y][0],160,use_threshold);
    fwrite((char *)&PackedBuf[0],sizeof(uchar) * len,1,fp);
    if (ferror(fp)) { status = FAILURE;break;}
    picMain[0].ScanLineDirectory[y][0] = Motorola16(len);
    picMain[0].ScanLineDirectory[y][1] =
    Motorola16((ushort)pic[0].scb[y]); blocklen += len;
}
if (status == FAILURE) { status = INVALID; break; }
rewind(fp); picMain[0].Length = Motorola32(blocklen);
fwrite(&picMain[0].Length,sizeof(PNTPIC),1,fp);
if (ferror(fp)) break; status = SUCCESS;
}
fclose(fp); PntFree();
if (status == SUCCESS) printf("Created: %s\n",pntfile);
else printf("Can't Create %s!\nOutput File Write Error!
Exiting...\n",pntfile);
return status;
}

```



## Transforming “raw” SHR Settings to APF File Settings

In the code above, some helper functions are called to transform the “raw” SHR data to the format expected by the APF file:

```
/* allocates memory and sets-up defaults for the type of
   PNT file output that will be produced */
sshort PntAlloc()
{
    sshort status = INVALID;
    uchar buf[10];
    if (PackAlloc(160) == INVALID) return status;

    switch(input_format) {
        case BROOKS_FMT:

            MultiPal = (MULTIPAL *) malloc(sizeof(MULTIPAL));
            if (NULL == MultiPal){
                PackFree();
                puts("Not Enough Memory for Multipalette... Exiting!");
                break;
            }
            brooksMain = (PNTBROOKS *) malloc(sizeof(PNTBROOKS));
            if (NULL == brooksMain) {
                puts("Not Enough Memory for Main... Exiting!");
                free(MultiPal); PackFree(); break;
            }
            /* set defaults and invariants */
            memset(&MultiPal[0].Length,0,sizeof(MULTIPAL));
            MultiPal[0].Length = Motorola32((ulong)sizeof(MULTIPAL));
            strcpy(&buf[1],"MULTIPAL"); buf[0] = 8;
            memcpy(&MultiPal[0].Kind[0],&buf[0],9);
            MultiPal[0].NumColorTables = Motorola16((ushort)200);
            /* The 200 Palettes will be assigned when the Brooks Palettes are built
            so nothing further to do except write to disk when done processing */
            memset(&brooksMain[0].Length,0,sizeof(PNTBROOKS));
            strcpy(&buf[1],"MAIN"); buf[0] = 4;
            memcpy(&brooksMain[0].Kind[0],&buf[0],5);
            brooksMain[0].PixelsPerScanline = Motorola16((ushort)320);
            /* as dumb as it may sound, the CiderPress file viewer needs one color
            table to be included in the MAIN block even when a Brooks MULTIPAL is
            used. */
            brooksMain[0].NumColorTables = Motorola16((ushort)1);
            brooksMain[0].NumScanLines = Motorola16((ushort)200);
            /* the scbs will be setup in the scan line directory with the packed
            lengths when the scanlines are packed... */
            status = SUCCESS; break;
    }
}
```

The code above sets-up the mode3200 APF and the code below sets-up the mode320 and mode640 APF.

```

case PIC_FMT:
default:
    picMain = (PNTPIC *) malloc(sizeof(PNTPIC));
    if (NULL == picMain) {
        puts("Not Enough Memory for Main... Exiting!");
        PackFree(); break;
    }
    /* set defaults and invariants */
    memset(&picMain[0].Length,0,sizeof(PNTPIC));
    strcpy(&buf[1],"MAIN"); buf[0] = 4;
    memcpy(&picMain[0].Kind[0],&buf[0],5);
    /* change to 640 if needed after loading raw file */
    picMain[0].PixelsPerScanline = Motorola16((ushort)320);
    picMain[0].NumColorTables = Motorola16((ushort)16);
    picMain[0].NumScanLines = Motorola16((ushort)200);
    status = SUCCESS; break;
}

return status;
}

```

### Little Endian and Big Endian Helper Functions

In the code above assignment of integral values to the APF headers is not done directly, but through the return of values. Whether we are writing or reading an APF, functions are provided to simplify indirect transformation to and from the APF's integral values.

The APF file format uses Motorola's Big Endian rather than INTEL's Little Endian storage format for 16 bit short integers and 32 bit long integers. If a compiler stores data internally in little endian, the byte order needs to be reversed to big endian, as in the case of the MinGW gcc compiler for Windows which I used to initially create all of this.

```

/* intel uses little endian */
/* motorola uses big endian */

/* for raw output */
ulong Intel32(ulong val)
{
    uchar buf[4];
    ulong *ptr;
    /* msb in largest address */
    buf[0] = (uchar) val &0xff; val >>=8;
    buf[1] = (uchar) val &0xff; val >>=8;
    buf[2] = (uchar) val &0xff; val >>=8;
    buf[3] = (uchar) val &0xff;
    /* cast back to unsigned long data type */
    ptr = (ulong *)&buf[0];
    val = ptr[0];
    return val;
}

```

```

/* for raw output */
ushort Intel16(ushort val)
{
    uchar buf[12];
    ushort *ptr;
    /* msb in largest address */
    buf[0] = (uchar) val &0xff; val >>=8;
    buf[1] = (uchar) val &0xff;
    /* cast back to unsigned short data type */
    ptr = (ushort *)&buf[0];
    val = ptr[0];
    return val;
}

/* for packed output */
ulong Motorola32(ulong val)
{
    uchar buf[4];
    ulong *ptr;
    /* msb in smallest address */
    buf[0] = (uchar) (val % 256); val = val/256;
    buf[1] = (uchar) (val % 256); val = val/256;
    buf[2] = (uchar) (val % 256); val = val/256;
    buf[3] = (uchar) (val % 256);
    /* cast back to unsigned long data type */
    ptr = (ulong *)&buf[0];
    val = ptr[0];
    return val;
}

/* for packed output */
ushort Motorola16(ushort val)
{
    uchar buf[2];
    ushort *ptr;
    /* msb in smallest address */
    buf[0] = (uchar) (val % 256); val = val/256;
    buf[1] = (uchar) (val % 256);
    /* cast back to unsigned short data type */
    ptr = (ushort *)&buf[0];
    val = ptr[0];
    return val;
}

```

P2p wants to work even if compiled on some old 68000 machine, so endianness needs to be observed when setting-up these defaults. While endianness may be arbitrary to a compiler's internal storage, it is not arbitrary when it comes to converting a PIC or Brooks file to an APF file.

## Brooks Palette to APF Palette Helper Function

In the preceding code for the PackApf() function, the first function call that is made when converting a Brooks file to a mode3200 APF file is a call to the helper function SetUpBrooksPalette(). Since all we are doing is reversing the order of the \$ORGB values during the copy to the MULTIPAL block, this is simply a matter of going forward in reverse for each of the 200 palettes (one for each scan-line) using an integer pointer.

```
void SetUpBrooksPalette()
{
  ushort *brookspal, *pntpal;
  sshort y,i,j;
  /* Brooks Palette Lines are in reverse order... the color value for
  color 15 is stored first.*/
  for (y=0;y<200;y++) {
    /* Build the Palette line for the APF from the Brooks Palette line */
    brookspal = (ushort *)&brooks[0].pal[y][0];
    pntpal = (ushort *)&MultiPal[0].ColorTableArray[y][0];
    /* According to Apple's Filetype Notes mode3200 palettes are in the
    same order as any other palette: 0..16 so flip the brooks palette end
    for end */
    for (i=0,j=15; i < 16; i++,j--) pntpal[i] = brookspal[j];
  }
}
```

Unlike mode320 and mode640 APF files, Mode3200 APF files do not store Brooks palettes in the MAIN information block, but in a separate block called a MULTIPAL.

Mode320 and mode640 PIC files represented as APF files are less convoluted than Brooks Files represented as APF files because they only have a single block; the entire PIC file is stored in the MAIN block. There is however some ambiguity even with PIC files converted to APF files because, typical of the mixed-mode tradition of the Apple II, the SHR display supports mixing mode640 and mode320 scan-lines on the screen at the same time. But APF files have the notion of something called a MasterMode which only allows for a single video mode which aims to compensate for the lack of a BytesPerScanline field by instead using a PixelsPerScanline and the MasterMode together to determine the number of bytes per scanline. I don't know whether mixed mode ever saw much use, so perhaps this isn't even an issue..

Brooks files do not share the same multiple mode disorder because mode640 is not supported by the format as far as I know, but are inflicted instead with a multiple palette disorder. The recommendation in Apple Computer's APF File Type Notes to provide a greyscale representation of a Brooks Image is almost as weird as describing the device dependent APF format as flexible. A Brooks Image of any complexity at all is unlikely to be able to use the same index throughout to represent the same color unless it is really a 16 color pic file that had a bad day and got lost in a mode3200 converter.

By definition, Brooks Palette Indexes, like most any index in palettized bitmapped graphics, do not have any relationship to the RGB values in the palette itself; therefore they can (and likely do) have a different index order and/or different colors for each of the 200 lines, unless colors are posterized and indices are sorted or unless Brooks format was mis-used to store a 16 color image. But, to satisfy the loaders that were written with this requirement in mind, p2p provides one (and only one) All Black "GreyScale" Color Table in the Main Block of the APF files it produces, created simply by memsetting the structure to NUL bytes when it is initialized.

By December 1991, when Apple Computer last revised the APF File Type Notes, device independent formats like BMP3 were in wide use, and other device independent formats had been around around for years. One wonders why Apple didn't use the opportunity of their final revision of the APF notes to replace the word "flexible" with something more accurate. Apple's standards seem more important at the start of a product life-cycle.

### The UnPackBytes() Function

```
/*
 * Unpack the Apple PackBytes format.
 *
 * Format is:
 * <flag><data> ...
 *
 * Flag values (first 6 bits of flag byte):
 * 00xxxxxx: (0-63) 1 to 64 bytes follow, all different
 * 01xxxxxx: (0-63) 1 to 64 repeats of next byte
 * 10xxxxxx: (0-63) 1 to 64 repeats of next 4 bytes
 * 11xxxxxx: (0-63) 1 to 64 repeats of next byte taken as 4 bytes
 *             (as in 10xxxxxx case)
 *
 * Pass the destination buffer in "dst", source buffer in "src", source
 * length in "srcLen", and expected sizes of output in "dstRem".
 *
 * Returns 0 on success, nonzero if the buffer is overfilled or
 * underfilled.
 */
```

```
/* the following code is taken literally from CiderPress
(ReformatBase.cpp) and dummied-down to C from C++. It is also lightly
modified to support a test mode when a NULL pointer is passed as a
destination buffer address.
```

```
this test mode is helpful for verifying files during decoding and also
for debugging.
```

```
if you decide to use this in your own programs Andy's licence (above)
is required and must be followed.
```

```
Sorry Andy. I wouldn't have been able to write a different decoder
unless I was decoding something different.
```

```
Apparently Out-encoding Apple Computer is easier done than Out-decoding
Andy McFadden. */
```

```

int UnPackBytes(uchar *dst, uchar * src, long dstRem, long srcLen)
{
uchar flag, val, valSet[4];
int count, i, unpacking = 1; /* active state = 1, test state = 0 */
if (NULL == dst) unpacking = 0;
while (srcLen > 0) {
    flag = *src++;
    count = (flag & 0x3f) + 1;
    srcLen--;
    switch (flag & 0xc0) {
    case 0x00:
        for (i = 0; i < count; i++) {
            if (srcLen == 0 || dstRem == 0) {
#ifdef DEBUG
printf("SHR unpack overrun1 (srcLen=%ld dstRem=%ld)\n", srcLen, dstRem);
#endif
                return INVALID;
            }
            if (unpacking) *dst++ = *src++;
            else *src++;
            srcLen--;
            dstRem--;
        }
        break;
    case 0x40:
        if (srcLen == 0) {
#ifdef DEBUG
printf("SHR unpack underrun2\n");
#endif
                return INVALID;
            }
            val = *src++;
            srcLen--;
            for (i = 0; i < count; i++) {
                if (dstRem == 0) {
#ifdef DEBUG
printf("SHR unpack overrun2 (srcLen=%d, i=%d of %d)\n", srcLen, i,
count);
#endif
                    return INVALID;
                }
                if (unpacking) *dst++ = val;
                dstRem--;
            }
            break;
    case 0x80:
        if (srcLen < 4) {
#ifdef DEBUG
printf("SHR unpack underrun3\n");
#endif
                return INVALID;
            }
            valSet[0] = *src++;
            valSet[1] = *src++;
            valSet[2] = *src++;
            valSet[3] = *src++;
            srcLen -= 4;

```

```

        for (i = 0; i < count; i++) {
            if (dstRem < 4) {
#ifdef DEBUG
printf("SHR unpack overrun3 (srcLen=%ld dstRem=%ld)\n",srcLen, dstRem);
#endif
                return INVALID;
            }
            if (unpacking) {
                *dst++ = valSet[0];
                *dst++ = valSet[1];
                *dst++ = valSet[2];
                *dst++ = valSet[3];
            }
            dstRem -= 4;
        }
        break;
case 0xc0:
    if (srcLen == 0) {
#ifdef DEBUG
printf("SHR unpack underrun4\n");
#endif
        return INVALID;
    }
    val = *src++;
    srcLen--;
    for (i = 0; i < count; i++) {
        if (dstRem < 4) {
#ifdef DEBUG
printf("SHR unpack overrun4 (srcLen=%ld dstRem=%ld count=%d)\n",srcLen,
dstRem, count);
#endif
            return INVALID;
        }
        if (unpacking) {
            *dst++ = val;*dst++ = val;*dst++ = val;*dst++ = val;
        }
        dstRem -= 4;
    }
    break;
default:
#ifdef DEBUG
    printf("Invalid Mask!\n");
#endif
    break;
    }
}
/* require that we completely fill the buffer */
if (dstRem != 0) {
#ifdef DEBUG
printf("SHR unpack dstRem at %d\n", dstRem);
printf("Flag = 0x%02x count = %d\n", flag &0xc0, count);
#endif
return INVALID;
}

return SUCCESS;
}

```

## **Decoding Packed SHR Files with UnPackBytes**

Decoding SHR files does not seem so mysterious as encoding them, from what I have seen. Now I could be wrong, and maybe somewhere there are a pile of PackBytes encoders that work on every modern platform, available with C language source code and freely distributed.

In which case, I am not very original. But at the risk of being more redundant than usual, and even less original, since Andy McFadden long ago provided PackBytes as open source with CiderPress, this section of the document provides the Reformatter from the p2p program, and some of the details of decoding Packed SHR files with UnPackBytes.

**/\* Similar logic to CiderPress's Reformat Handlers but comparatively just a subset since p2p converts screensize SHR images only for mode320, mode640 and mode3200.**

**Rules as to whether we want to handle these or not are based on similar logic to Andy's Logic but are more restrictive. But they ain't special either.**

**If you are into reading Andy's code, here's some equivalents:**

**Conversion from Packed to Raw:**

**ReformatPackedSHR - for conversion to PIC (SHR) - \$C1 \$0000**

**input\_format = PAK\_FMT**

**output\_format= PIC\_FMT**

**ReformatPacked3200SHR - for conversion to BROOKS (SH3) - \$C1 \$0002**

**input\_format = PAK\_FMT**

**output\_format= BROOKS\_FMT**

**ReformatAPFSHR - for conversion to PIC (SHR) or Brooks (SH3)**

**input\_format = PNT\_FMT**

**output\_format= PIC\_FMT or output\_format=BROOKS\_FMT**

**Conversion from Raw to Packed:**

**ReformatUnpackedSHR - for conversion to PNT \$C0 \$0002 or PAK \$C0 \$0001**

**Reformat3200SHR - for conversion to PNT \$C0 \$0002 or PA3 \$C0 \$0004**

**Additional Logic (there's no FT Note for this):**

**ReformatPacked3200SHR - No FTN for this and not included in CiderPress.**

**Andy handles a handful of oddball SHR files in CiderPress. I am not into that.**

**I could also have handled additional logic for SinglePalette 16 Color Packed files, simply based on a size comparison, but I really see no point; there needs to be some scope to all of this, so I am trying to stay with the files in the FTN's that are relatively standard. \*/**

**/\* if we want to handle the input file it is read into the input buffer previously allocated in main() and the input\_format is set to the input file type, a raw equivalent is dumped right in-here. It just makes more sense since we are rummaging through the darned things in memory anyway so they get half-decoded before we know what they are.**



It might look a little daunting but the logic is dead-dumb simple...

1. Unpack the Encoded File to a raw equivalent.
2. Write the raw equivalent to disk.

If the input file is invalid for whatever reason the `input_format` is set to invalid and this bad-boy returns. On completion of any file read operation `ReformatSHR()` closes the input file.

Any decoding needed is done, and while we're decoding the output file is written. If the input file is a raw file however, it is hived-off to a ex-process proxy decoder based on the Portable HackBytes output options: APF or PAK/PA3.

Whether decoding or encoding, `ReformatSHR` returns the `input_format` to `main()` and `main()` can decide what to do from here... based on the requested output.

I could have done more with this, but at this point converting from a packed APF to another packed file like PAK or PA3 seems to be overkill and likely not worth the effort, since a person can always unpack one and then repack to the third for romantic reasons if any.

These days, there's limited use for PackBytes encoded files, but likely even less use for APF files larger than a Hgs SHR display. But the point of the exercise was to write a more efficient PackBytes encoder, so having done so, evidence is needed, even if no other reason exists.

That's the real reason for providing a PAK file and its PA3 Brooks equivalent. Anything further is nice to see in a Hgs Paint Program if one could find more than only one that actually works.

\*/

```
short ReformatSHR(FILE *fp)
{
    unsigned int x,y;
    int status = SUCCESS;
    ulong flen, hlen;
    size_t packet;
    PNTHEAD *header;
    PNTPAL *pal;
    PNTSCB *scb;
    uchar *outbuf;
    ushort *brookspal, *pntpal, MinimumSize, NumScanLines;

    ushort offset, i, j, k;
    ushort MasterMode, NumColorTables, PixelsPerScanline, NumScanlines, NumPalettes;

    input_format = INVALID;

    /* get the length of the file */
    fseek(fp, 0L, 2);
    flen = ftell(fp);

    /* avoid empty files */
    if (flen < sizeof(PNTHEAD) || flen > 64000L) {
        fclose(fp);
        printf("Unsupported Input File Size = %ld. Exiting...\n",flen);
        return input_format;
    }
}
```

*/\* PackBytes - based on filesize alone*

**We don't have the luxury of a IIGs FileType and AuxType here. If someone wants to put garbage in, they will get garbage out. We aim to please.**

```
*/
if (output_format == PNT_FMT || output_format == PAK_FMT) {

    if (flen == PIC_LEN) {
        if (output_format == PAK_FMT) {
            if (use_tags == 0) strcat(pntfile, ".pak");
            else strcat(pntfile, ".pak#C00001");
        }
        else {
            if (use_tags == 0) strcat(pntfile, ".pnt");
            else strcat(pntfile, ".pnt#C00002");
        }
        input_format = PIC_FMT;
    }
    else if (flen == BROOKS_LEN) {
        if (output_format == PAK_FMT) {
            if (use_tags == 0) strcat(pntfile, ".pa3");
            else strcat(pntfile, ".pa3#C00004");
        }
        else {
            if (use_tags == 0) strcat(pntfile, ".pnt");
            else strcat(pntfile, ".pnt#C00002");
        }
        input_format = BROOKS_FMT;
    }
    else {
        fclose(fp);
        printf("Unsupported Raw Input File Size = %ld. Exiting...\n",flen);
        return input_format;
    }
}

/* read into the shr buffer */
rewind(fp);
packet = (size_t) (flen/4);
for (x = 0; x < flen; x+= packet) {
    fread((char *)&shrbuf[x],sizeof(char),packet,fp);
    if ((status = ferror(fp))!=0) break;
}
fclose(fp);
if (status!= SUCCESS) {
    printf("Input File Read Error: %d! Exiting...",status);
    input_format = INVALID;
}
return input_format;
}
```

What happens after the above code has executed is already listed in the HackBytes code. What follows is the unpacker which will decode files created by either Apple's PackBytes or our own HackBytes:

```

/* UnPackBytes */
if (flen == PIC_LEN || flen == BROOKS_LEN) {
    /* don't overwrite ourselves */
    fclose(fp);
    printf("Input File already unpacked! Exiting...\n");
    input_format = INVALID;
    return input_format;
}
rewind(fp);
packet = (size_t) (flen/4);
for (x = 0; x < flen; x+= packet) {
    fread((char *)&shrbuf[x],sizeof(char),packet,fp);
    if ((status = ferror(fp))!=0) break;
}

if (status == SUCCESS) {
    x = (unsigned) packet * 4;
    flen -= x;
    if (flen > 0) {
        packet = (size_t) flen;
        fread((char *)&shrbuf[x],sizeof(char),packet,fp);
        status = ferror(fp);
    }
    flen += x;
}
fclose(fp);

if (status != SUCCESS) {
    printf("Input File Read Error: %d! Exiting...",status);
    input_format = INVALID;
    return input_format;
}

```

After we check for some obvious errors we try to read the input file as an APF file first, then as an Eagle/PackBytes file. Our test for APF is quick so it deserves to be first; it includes a considerable amount of validation, not only because it is possible without unpacking the whole file like in the case of Eagle/Packbytes, but also because data coming out of an APF is what Apple Computer calls “flexible” so this lets us rule out the honest errors while we make the jokers who may have fed us a BMP or a pdf file just to see what happens wait until the very end. Anytime you can let the jokers wait until the end is a good time. While interpreting APFs is common sense, interpreting Apple Computer’s documentation for them, like much else to do with the Apple II, requires imagination and the reading of many once-expensive scraps to implement.

The code below is organized into a tight loop which breaks when the input file is unlikely to be an APF and which returns an error when the file is likely to be an APF in an unsupported variation. While not infallible it is certainly better than using file size which is what we need to do with all the other input formats that p2p handles.

```

/* when we are outputting to raw formats from pnt or pak we just overwrite a file if it already exists.
*/
/* do we want to handle this file? */
/* check for an APF file */
header = (PNTHEAD *)&shrbuf[0];

for (;;) {

/* check the MAIN information block. if the input file doesn't start with a MAIN block
assume it's not an APF file. */

/* 4 'M' 'A' 'I' 'N' */
if (header[0].Kind[0] != 4) break;

if (memcmp((char *)&header[0].Kind[1], "MAIN", 4) != 0) break;

/*
if (header[0].Kind[1] != 'M' || header[0].Kind[2] != 'A' ||
header[0].Kind[3] != 'I' || header[0].Kind[4] != 'N') break;

*/
/* at this point we know we have an APF file. */

/* The FTN for APF says that a ColorTable is not required, but goes on to say that at least one
GreyScale ColorTable is required to preview a mode3200 image with a MULTIPAL block.
*/
NumColorTables = Intel16(header[0].NumColorTables);
if (NumColorTables < 1 || NumColorTables > 16) {
/* CiderPress uses this criteria. Since I am using CiderPress as part of the validation suite for this
program and since this is consistent with the FTN's recommendation for a preview palette and since
DreamGraphix opens conversions from this program properly it seems reasonable to use the same
ColorTable criteria as CiderPress. */
printf("APF found with %u Color Tables!\n", NumColorTables);
puts("Valid range is 1-16! Exiting...");
break;
}

/* Check Horizontal Resolution */
/* MasterMode together with PixelsPerScanline determines the number of bytes to unpack per line.
Therefore the two must be set properly by their creator. */
MasterMode = Intel16(header[0].MasterMode);
PixelsPerScanline = Intel16(header[0].PixelsPerScanline);

if (PixelsPerScanline == 320 || PixelsPerScanline == 640) {
/* If MasterMode is 0x00 and PixelsPerScanline is 640 the APF is double the width of the screen.
If Mastermode is 0x80 and PixelsPerScanline is 320 the APF is half the width of the screen. */
if (MasterMode == 0 || MasterMode == 0x80) {
if ((MasterMode == 0 && PixelsPerScanline == 640) ||
(MasterMode == 0x80 && PixelsPerScanline == 320)) {
printf("APF found with MasterMode %x (hex) and PixelsPerScanline %d!\n",
MasterMode, PixelsPerScanline);
puts("Valid range is 0 (hex) and 320 or 80 (hex) and 640! Exiting...");
break;
}
}
}
}

```

```

else {
    /* if the MasterMode has not been set properly the file is broken and can't be trusted */
    printf("APF found with MasterMode %x (hex)!\n",MasterMode);
    puts("Valid range is 0 or 80 (hex)! Exiting...");
    break;
}
}
else {
    /* if the APF is not a screenwidth APF we don't handle it because we are converting to full-screen
    RAW formats in this program. */
    printf("APF found with %u PixelsPerScanline!\n",PixelsPerScanline);
    puts("Supported Horizontal Resolutions are 320 or 640! Exiting...");
    break;
}

offset = (ushort) (sizeof(PNTHEAD) + (NumColorTables * 32));
scb = (PNTSCB *)&shrbuf[offset];
NumScanLines = Intel16(scb[0].NumScanLines);
if (NumScanLines != 200) {
    printf("APF found with %u ScanLines!\n",NumScanLines);
    puts("Supported Vertical Resolution is 200! Exiting...");
    break;
}

/* the minimum size for the MAIN block of a screen width APF adds the fixed fields together
with the ColorTables (32 * NumColorTables) and (200 lines x 4 bytes).
4 bytes is the minimum number of bytes for a one colored PackBytes encoded line of 160 bytes */
MinimumSize = (ushort) (sizeof(PNTHEAD) + (NumColorTables * 32) + sizeof(PNTSCB) + 800);
hlen = Intel32(header[0].Length);
if (hlen < MinimumSize) {
    printf("APF MAIN Block Length %ld below Minimum Size %u!\n",hlen,MinimumSize);
    puts("Exiting...");
    break;
}
/* the MAIN block will never be longer than the file size */
if (hlen > flen) {
    printf("APF MAIN Block Length %ld exceeds File Size %ld!\n", hlen, flen);
    puts("Exiting...");
    break;
}

/* I am expecting the MULTIPAL block with 200 sequential ColorTables to directly follow the MAIN
Block in a mode3200 file. If somebody is using a NOTES block or some other Block we will not
handle the file as a mode3200 file.

If somebody is using a MULTIPAL instead of the ColorTable in the MAIN Block for a mode320 or a
mode640 file we will just use the ColorTable in the Main Block.

We are not prepared to handle every funky variation like Andy does.

So unless a Brooks file has been properly encoded in the APF it may end-up with pieces missing, or it
could end-up as a PIC file with scb's that don't make sense. */

for (;;) {
    output_format = PIC_FMT;

```

```

    NumPalettes = 0;
    if (PixelsPerScanline != 320) break;

    /* if a MultiPalette is used for other than a mode3200 file I have no idea what this might be about */
    MultiPal = (MULTIPAL *)&shrbuf[hlen];
    if (MultiPal[0].Kind[0] != 8) break;
    if (memcmp((char *)&MultiPal[0].Kind[1], "MULTIPAL", 8) != 0) break;
    /* we have a MultiPalette */
    NumPalettes = Intel16(MultiPal[0].NumColorTables);
    if (NumPalettes == 200) output_format = BROOKS_FMT;
    break;
}

if (output_format == BROOKS_FMT) {
    if (NULL == (outbuf = malloc(38400))) {
        puts("No memory for Write Buffer! Exiting...");
        input_format = INVALID;
        return input_format;
    }
    memset(&outbuf[0], 0, 38400);
    brooks = (BROOKSFILE *)&outbuf[0];

    for (y=0; y<200; y++) {
        /* Build the Palette line for the Brooks File from the APF Palette line */
        brookspal = (ushort *)&brooks[0].pal[y][0];
        pntpal = (ushort *)&MultiPal[0].ColorTableArray[y][0];
        /* According to Apple's Filetype Notes mode3200 palettes are in the same order as any other palette:
        0..16 so flip the APF palette end for end */
        for (i=0, j=15; i < 16; i++, j--) brookspal[j] = brookspal[j] = pntpal[i];
    }

    if (use_tags == 0) strcat(pntfile, ".sh3");
    else strcat(pntfile, ".sh3#C10002");
}
else {
    if (NULL == (outbuf = malloc(32768))) {
        puts("No memory for Write Buffer! Exiting...");
        input_format = INVALID;
        return input_format;
    }
    memset(&outbuf[0], 0, 32768);
    offset = (ushort) sizeof(PNTHHEAD);
    pal = (PNTPAL *)&shrbuf[offset];
    pic = (PICFILE *)&outbuf[0];

    /* copy palette from APF file to picfile */
    memcpy(&pic[0].pal[0][0], &pal[0].ColorTable[0][0], (NumColorTables * 32));

    /* build the scbs from the APF file */
    for (y=0; y<200; y++) {
        pic[0].scb[y] = (uchar) Intel16(scb[0].ScanLineDirectory[y][1]);
    }
    if (use_tags == 0) strcat(pntfile, ".shr");
    else strcat(pntfile, ".shr#C10000");
}
}

```

```

fp = fopen(pntfile,"wb");
if (NULL == fp) {
    free(outbuf);
    printf("Unable to open %s for output!\nExiting...\n",pntfile);
    input_format = INVALID;
    return input_format;
}

```

We are committed to writing an APF by the time we get to here. The only thing that can stop us now is a file write error:

```

/* now unpack the bytes - packed scan-lines are below the APF header info */
offset = (ushort) (sizeof(PNTHEAD) + (NumColorTables * 32) + sizeof(PNTSCB));
/* length of packed scanlines */
hlen -= offset;

if(UnPackBytes((uchar *)&outbuf[0],(uchar *)&shrbuf[offset],(long)32000, (long)hlen) != INVALID)
{
    input_format = PNT_FMT;
    fwrite((uchar *)&outbuf[0],sizeof(uchar) * 32000,1,fp);
    if (ferror(fp)) input_format = INVALID;
    else {
        if (output_format == BROOKS_FMT) fwrite((uchar *)&outbuf[32000],
            sizeof(uchar) * 6400,1,fp);
        else fwrite((uchar *)&outbuf[32000],sizeof(uchar) * 768,1,fp);
        if (ferror(fp)) input_format = INVALID;
    }
}
if (input_format == INVALID) printf("Can't Create %s!\nOutput File Write Error!
Exiting...\n",pntfile);
}
else {
    printf("Can't Create %s!\nUnPackBytes Error! Exiting...\n",pntfile);
}

fclose(fp);
if (input_format == INVALID) remove(pntfile);
else printf("Created: %s\n",pntfile);

free(outbuf);
return input_format;
}

```

If we have got to this point, whatever input file we have is either an Eagle/PackBytes file of some kind or can safely be considered garbage shoved in here by some joker. If the input file decodes either to the size of a Brooks file or the size of a Pic file we proceed on faith alone that it is in fact a file that we want to handle. This where we do a dry-run first to save a few cycles while we determine if we can unpack cleanly to the size of the larger Brooks file. If the larger file is not a clean unpack we try to unpack to the size of the smaller Pic file. If that fails, we give-up and return an error and hopefully the joker who is feeding us garbage eventually gives-up trying to break us:

```

/* two more tests - first try to decode a packed Brooks File */
if(UnPackBytes(NULL,(uchar *)&shrbuf[0],(long)BROOKS_LEN, (long)flen) != INVALID) {
    input_format = PAK_FMT;
    output_format = BROOKS_FMT;
    if (NULL == (outbuf = malloc(38400))) {
        puts("No memory for Write Buffer! Exiting...");
        input_format = INVALID;
        return input_format;
    }

    if (use_tags == 0) strcat(pntfile, ".sh3");
    else strcat(pntfile, ".sh3#C10002");

    fp = fopen(pntfile, "wb");
    if (NULL == fp) {
        free(outbuf);
        printf("Unable to open %s for output!\nExiting...\n", pntfile);
        input_format = INVALID;
        return input_format;
    }
    UnPackBytes((uchar *)&outbuf[0],(uchar *)&shrbuf[0],(long)BROOKS_LEN, (long)flen);
    fwrite((uchar *)&outbuf[0],sizeof(uchar) * 32000,1,fp);
    if (ferror(fp)) input_format = INVALID;
    else {
        fwrite((uchar *)&outbuf[32000],sizeof(uchar) * 6400,1,fp);
        if (ferror(fp)) input_format = INVALID;
    }
    fclose(fp);
    if (input_format == INVALID) {
        printf("Can't Create %s!\nOutput File Write Error! Exiting...\n", pntfile);
        remove(pntfile);
    }
    else {
        printf("Created: %s\n", pntfile);
    }
    free(outbuf);
    return input_format;
}
else {
    /* if it's not a packed Brooks File try to decode a Packed PIC File */
    if(UnPackBytes(NULL,(uchar *)&shrbuf[0],(long)PIC_LEN, (long)flen) != INVALID) {
        input_format = PAK_FMT;
        output_format = PIC_FMT;
        if (NULL == (outbuf = malloc(32768))) {
            puts("No memory for Write Buffer! Exiting...");
            input_format = INVALID;
            return input_format;
        }

        if (use_tags == 0) strcat(pntfile, ".shr");
        else strcat(pntfile, ".shr#C10000");
    }
}

```



```

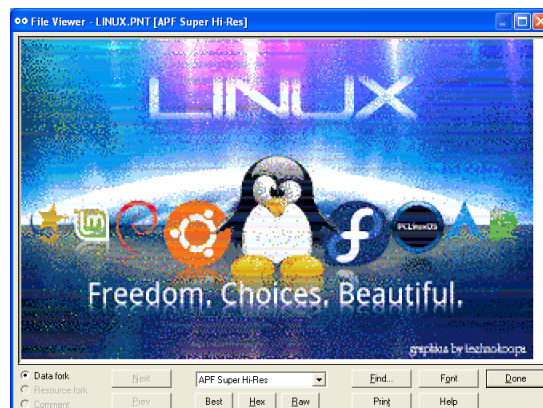
fp = fopen(pntfile,"wb");
if (NULL == fp) {
    free(outbuf);
    printf("Unable to open %s for output!\nExiting...\n",pntfile);
    input_format = INVALID; return input_format;
}

UnPackBytes((uchar *)&outbuf[0],(uchar *)&shrbuf[0],(long)PIC_LEN, (long)flen);
fwrite((uchar *)&outbuf[0],sizeof(uchar) * 32000,1,fp);
if (ferror(fp)) input_format = INVALID;
else {
    fwrite((uchar *)&outbuf[32000],sizeof(uchar) * 768,1,fp);
    if (ferror(fp)) input_format = INVALID;
}
fclose(fp);
if (input_format == INVALID) {
    printf("Can't Create %s!\nOutput File Write Error! Exiting...\n",pntfile);
    remove(pntfile);
}
else {
    printf("Created: %s\n",pntfile);
}
free(outbuf);
return input_format;
}
else {
    printf("Unsupported Packed Input File!\nNot a Packed PIC or Brooks File. Exiting...\n");
}
}

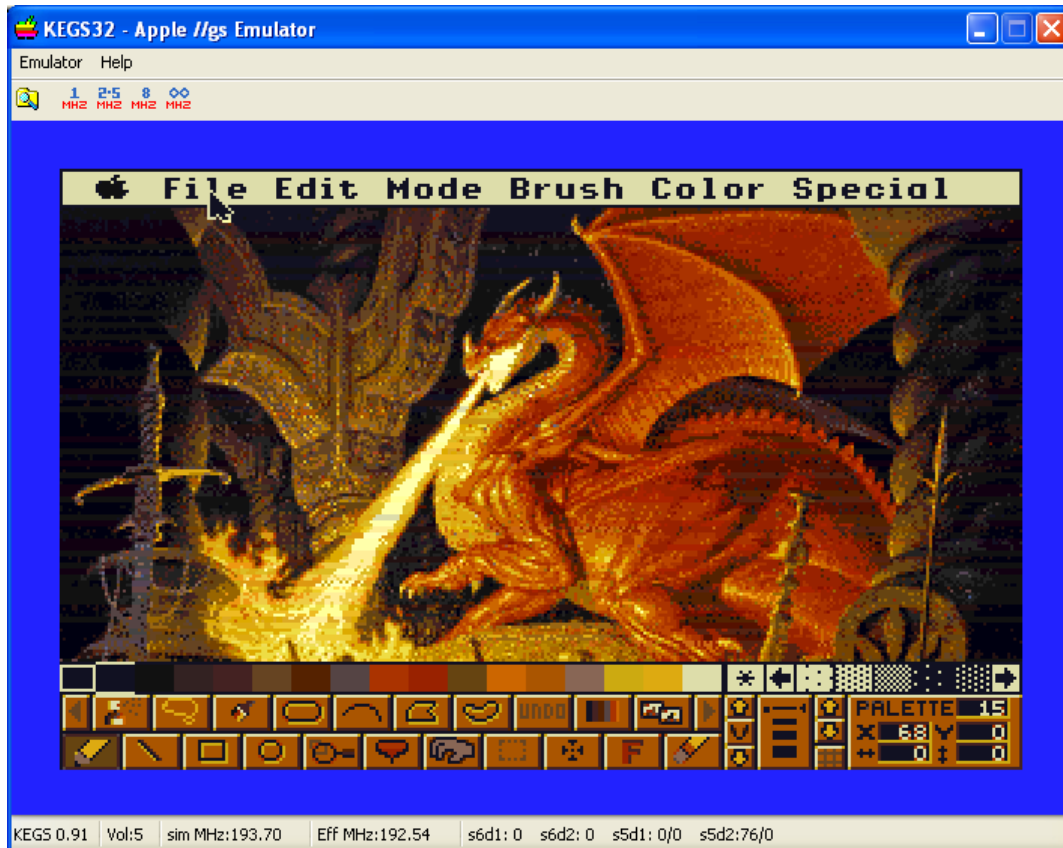
return input_format;
}

```

So that is how we handle reading and unpacking and writing the “raw” SHR files that we learned to create using Portable HackBytes. After all, only a brain-dead messed-up dysfunctional jerk of a PackBytes decoder would care whether an SHR file was created on an Apple Iigs Computer or if we created and encoded the file on an iPhone, any more than if a GIF or a ZIP file was created in Linux and loaded in Windows or OSX.



## Test Results - Compatibility And Regression

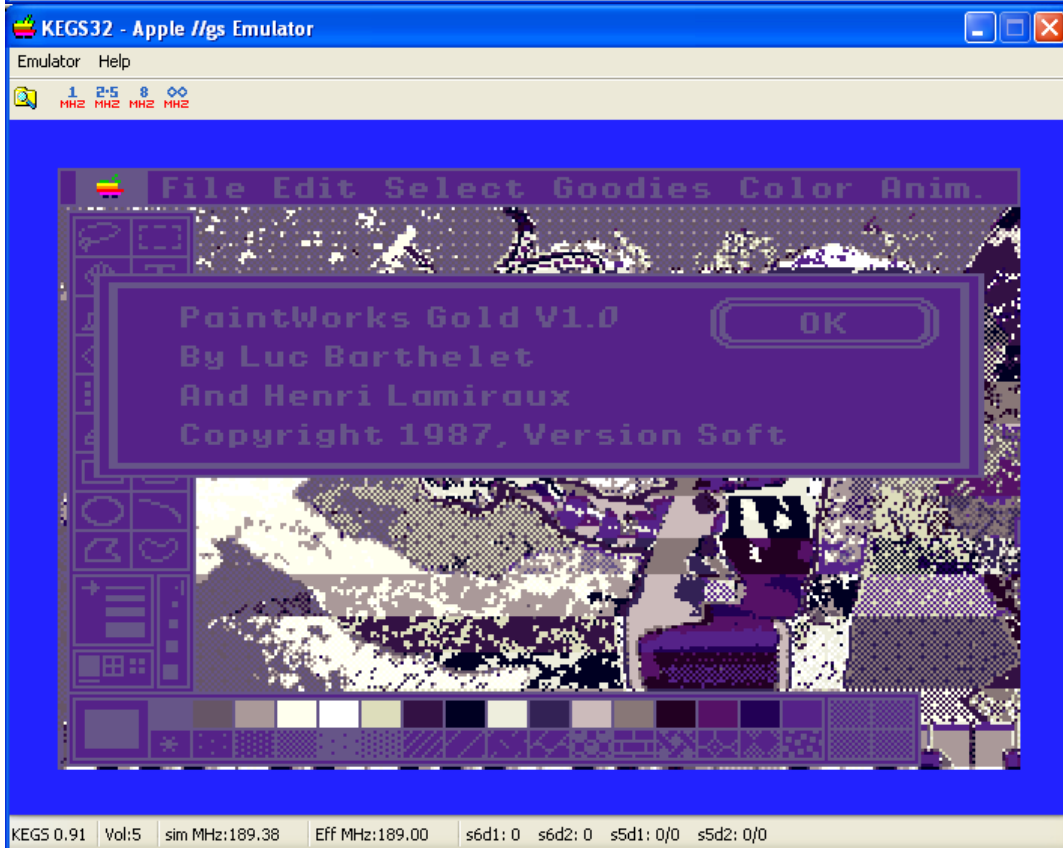
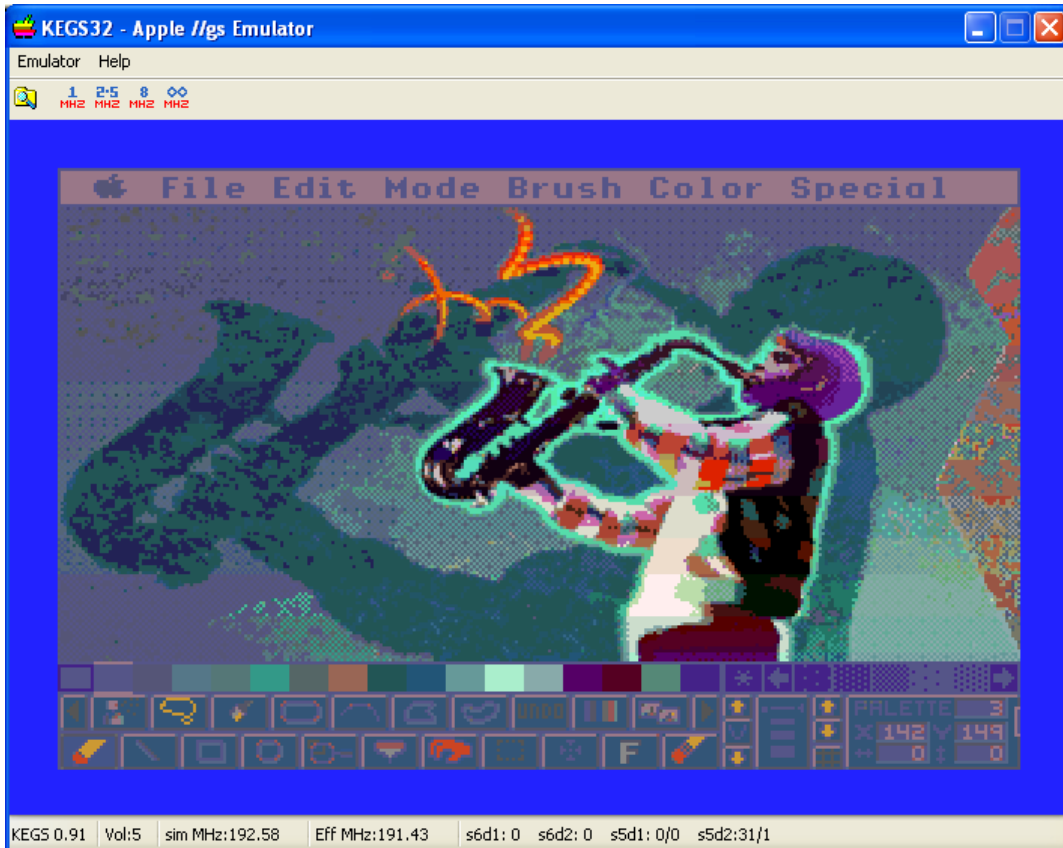


Portable HackBytes has seen a considerable amount of testing for regression and equivalence using a small test suite of programs which, in addition to my own 8 bit APF loader written in Aztec C65, include the CiderPress File Viewer, Ron Mercer's SHR View, DreamGraphix™ and PaintWorks Gold.

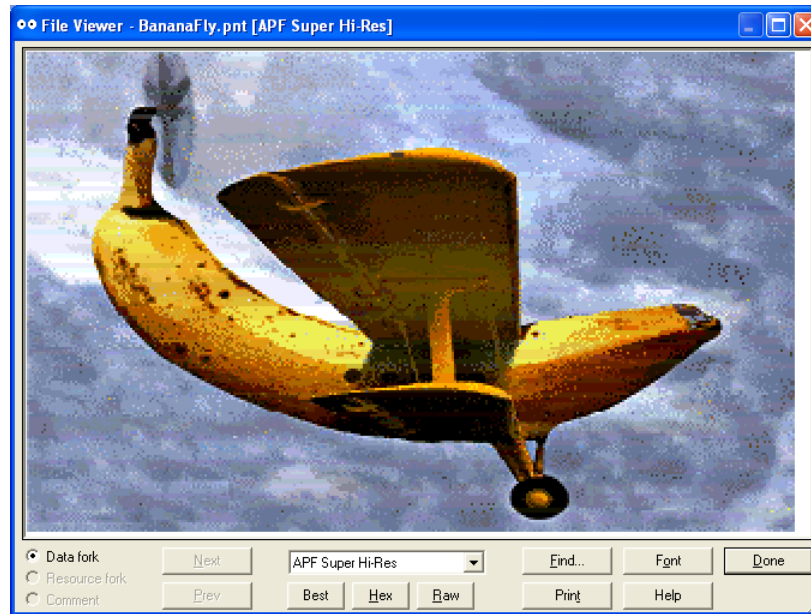
The CiderPress File Viewer, SHR View and DreamGraphix properly unpack and displayed both APF and Eagle/PackBytes PNT files created by p2p using Portable HackBytes. PaintWorks Gold also has no problems unpacking APF images compressed with Portable HackBytes.

But PaintWorks Gold seems incapable of displaying even an uncompressed raw pic file properly, so offers little value as a testing tool except to confirm the compatibility of Portable HackBytes compressed APF images created by p2p. PaintWorks Gold also does not recognize files saved in the Eagle/Packbytes format at all, making it an even poorer choice as a testing tool, and perhaps a poor choice as a Paint Program.

*The image below converted to an APF file using p2p and Portable HackBytes decodes and displays properly in DreamGraphix, and decodes properly in PaintWorks Gold.*



## Test Results - Performance – Comparing Apples to Bananas



A complex mode320 or mode640 SHR file with few repeats and every color used probably won't compress very well using PackBytes. A complex mode3200 Brooks file is generally even larger because it contains 200 palettes instead of the 16 required for mode320 and mode640. It may not even be worthwhile to compress complex screen size SHR files, considering the extra programming overhead required to unpack and display these. However, for a mode3200 file extra programming is always required. But whether to pack them or leave them raw depends on what you want them for.

### Test Results - Mode3200 Files

	<b>Raw</b>	<b>Blocks</b>	<b>PAK</b>	<b>Blocks</b>	<b>APF</b>	<b>Blocks</b>
Artemus.sh3	38,400	77	36,996	74	37,972	76
Boating.sh3	38,400	77	37,174	74	38,065	76
Colorseum.sh3	38,400	77	32,741	65	33,866	68
Galette.sh3	38,400	77	38,389	76	39,278	78
Pyramid.sh3	38,400	77	36,292	72	37,168	74
Reclining.sh3	38,400	77	37,458	75	38,349	76
Sphinx.sh3	38,400	77	36,068	72	37,587	75
<b>Total</b>	<b>268,800</b>	<b>539</b>	<b>255,118</b>	<b>508</b>	<b>262,285</b>	<b>523</b>

While not altogether meaningless, the table above does not accurately reflect either the quality of the raw SHR file, or the methods used to create the raw SHR file. Since I have selected "raw" SHR files above that have many colors, created with my general purpose BMP2SHR utility, which makes no attempt to organize or order palettes at the image level, this establishes a pretty accurate worst case compression ratio scenario.

## **The Contenders - Not Your Father's SHR Converter**

STYNX has developed a highly optimal “Modern” SHR converter, using the ImageMagick API, which, among other things, organizes palettes for re-use throughout an image; this is not your Father's SHR converter! His converter is capable of advanced techniques such as random dithering which are beyond the capabilities of BMP2SHR (my SHR converter).

On the other hand, BMP2SHR uses “Classic” population method quantization to map a palette. Each line's palette in BMP2SHR's mode3200 conversion can be quite differently ordered and differently colored from the other lines. BMP2SHR is also quite faithful about using 16 original colors rather than worrying about color space theories.

STYNX's palettes are organized in potentially re-usable segments leading to more potential repeats, and more consistent coloring between lines but more posterization (and rings) when dithers are not used. Without discussing both converters in detail, it is impossible to compare all the differences between the two. BMP2SHR is reasonably capable, but usually more subject to banding. STYNX's converter generally does a better job since it has dithers to help it along, but even without dithers it does a very good job of posterizing. I have disabled STYNX's dithers for my comparison.

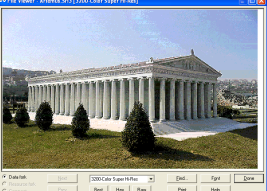

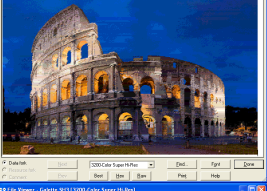


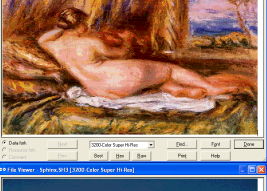

Both converters handle 24 bit TrueColor images. Like p2p and Portable HackBytes, they run on Modern Computers and not on the IIgs, and literally convert in less than a second. Comparing either's latency to something on the IIgs is irrelevant.

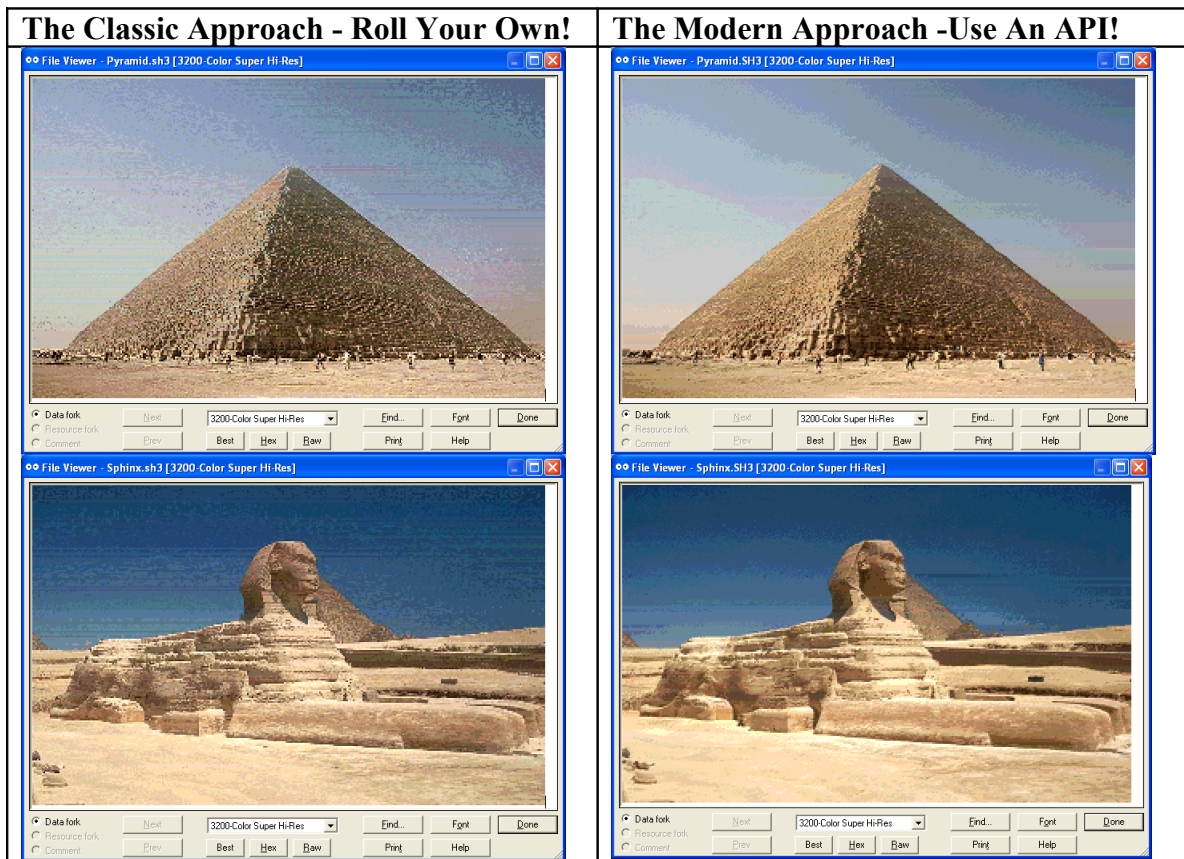
But for the purposes of this discussion, we are concerned with a comparison between packing “raw” SHR images of “Classic” quality versus “Modern” quality and not processing speed. I could have used some other converters, but since these are the two SHR converters I've had direct involvement with (although STYNX dreamed-up his converter on his own), I am exercising my author's prerogative of using my own converter as a worst case scenario for file size, and disabling some of STYNX's advanced features for a more objective cross-reference and meaningful user experience.

SHR Converters may not all be created equal, but if I had used STYNX's dithering features and the other advanced techniques that it supports, we would have seen better looking images from ImageMagick. I do apologize to Jonas for crippling his converter for this example, but I needed to. Better quality images achieved by dithering would also have resulted in more complex SHR images from STYNX's converter that would have compressed less efficiently, so we wouldn't have had much difference to compare.

## **Test Results – Mode320 Files**

The table below compares Portable HackBytes between a “Classic” conversion and “Modern” conversion.

	Raw	Blocks	BMP2SHR PackBytes	Blocks	STYNX PackBytes	Blocks
	38,400	77	36,996	74	35,742	71
	38,400	77	37,174	74	37,710	75
	38,400	77	32,741	65	31,498	63
	38,400	77	38,389	76	38,379	76
	38,400	77	36,292	72	30,029	60
	38,400	77	37,458 97.54%	75 97.4%	37,335 97.22%	74 96.1%
	38,400	77	36,068 93.92%	72 93.51%	32,616 84.94%	65 84.42%
<b>Total</b>	<b>268,800</b>	<b>539</b>	<b>255,118</b>	<b>508</b>	<b>243,309</b>	<b>484</b>
<b>Compression Ratio</b>	<b>100%</b>	<b>100%</b>	<b>94.90%</b>	<b>94.25%</b>	<b>90.51%</b>	<b>89.80%</b>



## Comparing Apples to Apples

Portable HackBytes generally performs somewhat better than Apple Computer's PackBytes as near as I can tell.

Based on a series of tests of a set of 19 files in the Eagle/PackBytes format, files compressed using Apple Computer's native PackBytes used an average of more than 2% disk space (in ProDOS 512K blocks) than the same SHR files compressed using Portable HackBytes, and although it is probably less important than the actual space used on a ProDOS disk, overall the natively compressed files were around 3% larger based on file size.

While this difference of 15 ProDOS blocks is not a significant size difference, imagine what you might have said if it were the wrong way around and the files produced by Portable HackBytes were 15 blocks larger than the files produced by Apple Computer's routines. Since it only took about a second to encode all 19 files on my modern computer, and the files were smaller when I was done, and survived my compatibility tests it seems pointless to use the Apple IIgs PackBytes routines at all if one has a modern computer and takes the Classic Approach instead of taking the Modern approach of using the Apple IIgs API on a Classic computer.

	Portable HackBytes()	Blocks	Apple IIgs PackBytes	Blocks	Apple IIgs Bloat %
ANGELFISH.pak	9,982	21	10,296	22	4.76
ASTRONUT.pak	26,315	53	26,375	53	0
BEHEMOTH.pak	15,484	32	15,797	32	0
BIG.pak	5,167	12	5,589	12	0
BUTTERFLY.pak	10,391	22	10,500	22	0
CD.pak	10,814	23	11,236	23	0
CLOWN.pak	23,656	48	24,135	49	2.08
COGITO.pak	22,340	45	22,842	46	2.22
COTTAGE.pak	21,165	43	21,571	44	2.33
FIGHTERS.pak	15,287	31	15,635	32	3.23
FLOWER.pak	17,469	36	18,584	38	5.56
JAZZ.pak	13,216	27	14,285	29	7.41
KNIFE.pak	22,640	46	23,153	47	2.17
LORI.pak	25,860	52	26,166	53	1.92
MAX.pak	11,943	25	13,219	27	8.00
OWL.pak	22,699	46	22,741	46	0
RED.DRAGON.pak	25,017	50	25,524	51	2.00
TAJ.pak	22,983	46	23,288	47	2.17
TUT.pak	15,585	32	15,848	32	0
<b>Total</b>	<b>338,013</b>	<b>690</b>	<b>346,784</b>	<b>705</b>	<b>2.17</b>

